REACT JS – HANDS ON / DESCRIPTIVE ANSWERING

1. How to create components in React?

   Components in React can be created using two approaches:

   - **Function Component:** A JavaScript function that returns JSX

```
function MyComponent() {
  return <div>Hello World!</div>;
}
```

   - **Class Component:** A JavaScript class that extends React.Component and includes a render() method returning JSX.

```
class MyComponent extends React.Component {
  render() {
    return <div>Hello World!</div>;
  }
}
```

2. When to use a Class Component over a Function Component?

   - You need to manage state using useState or useReducer.
   - You need to use lifecycle methods like componentDidMount, componentDidUpdate, or componentWillUnmount.
   - You need to use context API to share data between components.
   - You need to use refs to access DOM elements directly.
   - You are working with legacy code that uses class components.

3. What are Pure Components and explain with a snippet?

   - Pure Components in React are components that do not re-render if their props and state haven't changed. They implement a shallow comparison of props and state.

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

class MyPureComponent extends PureComponent {
  render() {
    return <div>{this.props.name}</div>;
  }
}
```

4. What is the diUerence between state and props?

- **State:** Managed within the component and can change over time. Used to handle dynamic data.
- **Props:** Passed to the component from its parent. They are immutable and allow data to flow between components.

```jsx
function ChildComponent({ name }) {
  return <div>{name}</div>;
}
```

5. What is the use of react-DOM package and name its packages?

react-DOM provides methods to interact with the DOM in web applications:

- render(): Renders React elements into the DOM.
- hydrate(): Hydrates server-rendered HTML into a React app.
- unmountComponentAtNode(): Removes a mounted React component.
- createPortal(): Renders children into a different part of the DOM.

6. Write a recursive program of your choice using recursive component

Here's a recursive component that renders a list of nested comments:

```jsx
function Comment({ comment }) {
  return (
    <div>
      <p>{comment.text}</p>
      {comment.replies && (
        <div style={{ marginLeft: '20px' }}>
          {comment.replies.map((reply, index) => (
            <Comment key={index} comment={reply} />
          ))}
        </div>
      )}
    </div>
  );
}
```

The given code defines a **recursive React component** called Comment, which renders nested comments using recursion.

1. **Base Component**: The Comment component receives a comment prop, which contains the text of the comment and potential replies.
2. **Rendering Comment Text**: The component first renders the main comment's text within a <p> tag.
3. **Recursive Rendering**: If the comment object contains a replies property, the component will render each reply by calling the same Comment component recursively for each nested comment.
4. **Indentation**: Replies are rendered with increased left margin, creating a visual indentation to represent the nested structure.
5. **Recursive Nature**: This recursive structure allows the component to render an arbitrary depth of nested comments without having to explicitly define each level. The recursion ends when there are no more replies to render.

Example:

If a comment has nested replies, the Comment component calls itself to render each reply in the same format. This is a classic use of recursion to handle hierarchical data in UI, such as nested comments.

7. How to call an API from react JS, a sample function call to prove the same

You can call an API using fetch or libraries like axios. Here's an example using fetch:

```
function App() {
  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/posts')
      .then(response => response.json())
      .then(data => console.log(data));
  }, []);

  return <div>API Data fetched in console</div>;
}
```

To call an API in React, you can use the fetch method or libraries like Axios. In the provided example, the fetch function is used to make a GET request.

1. **useEffect Hook**: The API call is made inside the useEffect hook, which ensures the request runs only once when the component mounts.
2. **Fetching Data**: The fetch function sends a GET request to the provided URL (https://jsonplaceholder.typicode.com/posts). It returns a promise, which resolves to the response object.
3. **Parsing Response**: The response.json() method converts the response to a JSON format, which is handled in the subsequent .then() block.
4. **Logging Data**: The fetched data is logged to the console.
5. **Rendering**: The component renders a simple message, while the API data is processed in the background.

This is a basic example of how to fetch and handle API data in React.

8. Create a react JS function to iterate and list the files of the project structure of react APP.

You can simulate file listing by iterating over an array of file names:

- In this example, the **FileList** component takes an array of file names as a prop (files). It then uses the **.map()** method to iterate over the array, rendering each file name inside a <li> element. The component outputs the list of files as a series of <li> elements inside an unordered list (<ul>).
- The array ['App.js', 'index.js', 'style.css'] is passed as the files prop when rendering the FileList component. This simulates listing files in a React project structure.

```
function FileList({ files }) {
  return (
    <ul>
      {files.map((file, index) => (
        <li key={index}>{file}</li>
      ))}
    </ul>
  );
}
```

Pass the files array as a prop:

```
<FileList files={['App.js', 'index.js', 'style.css']} />
```

9. Create a calculate APP using react JS components + HTML

- To create a simple calculator app using React JS, you can follow this example. It includes basic arithmetic operations (addition, subtraction, multiplication, and division) and uses functional components, state, and event handling.

Steps:

1. Setup the React Environment:

Before starting the project, you'll need to set up a React environment. This can be done by running npx create-react-app calculator-app from the terminal, which will scaffold a basic React project.

2. Break Down the Calculator's Functionality:

The goal is to create a calculator that can handle simple arithmetic operations: addition, subtraction, multiplication, and division. The app will have buttons for digits (0–9) and arithmetic operators (+, -, *, /), as well as buttons for clearing the input and calculating the result.

3. Structure the Calculator as React Components:

React is based on the concept of breaking down the UI into components. For this app, you'll mainly need the following components:

- **Display Component**: This will show the current input and result.
- **Button Component**: For each button (digits and operations).
- **Calculator Component**: This will wrap everything together and manage the state.

4. Using Hooks for State Management:

React's useState hook will be used to manage the calculator's input and result. You will need to store the current input string (as users press the buttons) and the final result after they press the "equals" button.

- The **input state** will hold the current input (e.g., "23+5").

- The **result state** will hold the final calculation when the user presses the "=" button.

5. Button Interaction & Event Handling:

Each button will have an onClick handler that updates the input state. When the user clicks a digit or operator, that value gets appended to the input. There will be special buttons for clearing the input (resetting the state) and for performing the calculation (evaluating the input and updating the result).

6. Performing Calculations:

The calculator will use JavaScript's built-in eval() function to evaluate the string expression entered by the user. This is a simple way to perform arithmetic calculations based on the input string, though in a production environment, a safer alternative would be preferable to avoid potential security issues.

7. Styling the Calculator:

You can add CSS to make the calculator visually appealing. This might include styling for the display area, buttons, and general layout. CSS classes can be used to define things like button size, hover effects, and positioning.

8. Handling Edge Cases:

It's important to handle potential errors, such as trying to divide by zero or entering invalid sequences (like multiple operators in a row). You can wrap the evaluation logic in a try-catch block to handle such cases gracefully and display an error message to the user when necessary.

9. Rendering the Calculator:

Finally, the calculator will be rendered inside the main App component. The App component will import and display the Calculator component, which in turn renders the display and buttons.

10. Change any CSS style dynamically on click of Button using state, and react fragment to render dynamic HTML

To dynamically change CSS styles on the click of a button in React, you can use the useState hook to track the current style and React.Fragment to render elements without adding unnecessary nodes.

1. **State Management**: You use useState to store a boolean that indicates whether the component should use the default or the alternate CSS style.
2. **Styling Logic**: Define two sets of styles—one for the default view and one for the alternate view (e.g., different background colors or text colors). These styles can be applied inline using the style attribute in React.
3. **Button Click Handler**: On each button click, toggle the boolean state value. This re-renders the component and applies the correct style based on the updated state.
4. **React.Fragment**: Use React.Fragment to group elements like the div containing content and the button without adding extra HTML elements.
5. **Dynamic Update**: The button can also dynamically change its label to indicate the current style status (e.g., "Switch to Default Style" or "Switch to Changed Style").

When the user clicks the button, the style of the element changes instantly, creating a dynamic, interactive UI.