1. Overview of Component Lifecycle

The React component lifecycle consists of different phases that a component goes through, from initialization to unmounting. Each phase has specific lifecycle methods associated with it.

Diagram - Component Lifecycle

```
Initialization
      |
Mounting Phase
      |
Updating Phase
      |
Unmounting Phase
```

These methods allow you to perform specific actions when the component is created, updated, or destroyed. Let's go through the different lifecycle methods with code snippets and examples:

Here's the correct order of the most commonly used lifecycle methods:

- constructor(props)
- static getDerivedStateFromProps(props, state)
- render()
- componentDidMount()
- componentDidUpdate(prevProps, prevState)
- shouldComponentUpdate(nextProps, nextState)
- componentWillUnmount()

Different Lifecycle methods

1. constructor(props): The constructor is the first method called when a component is created. It is used to initialize state and bind event handlers. Here's an example:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

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

2. componentDidMount(): The componentDidMount method is invoked immediately after the component is mounted (inserted into the DOM tree). It is commonly used to make API calls or initialize third-party libraries. Here's an example:

```
class MyComponent extends React.Component {
  componentDidMount() {
    // Make an API call or initialize a library
    fetchData().then(data => {
      this.setState({ data });
    });
  }

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

3. componentDidUpdate(prevProps, prevState): The componentDidUpdate method is called after the component is updated. It is useful for performing side effects when props or state change. Make sure to include a condition to prevent an infinite loop when updating state inside this method. Here's an example:

```
class MyComponent extends React.Component {
  componentDidUpdate(prevProps, prevState) {
    if (this.state.count !== prevState.count) {
      console.log('Count updated:', this.state.count);
    }
  }

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

4. shouldComponentUpdate(nextProps, nextState): The shouldComponentUpdate method is invoked before the component is updated. It allows you to control whether the component should re-render or not. Returning false from this method will prevent the component from updating. Here's an example:

```
class MyComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    // Only update if the count is odd
    return nextState.count % 2 === 1;
  }

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

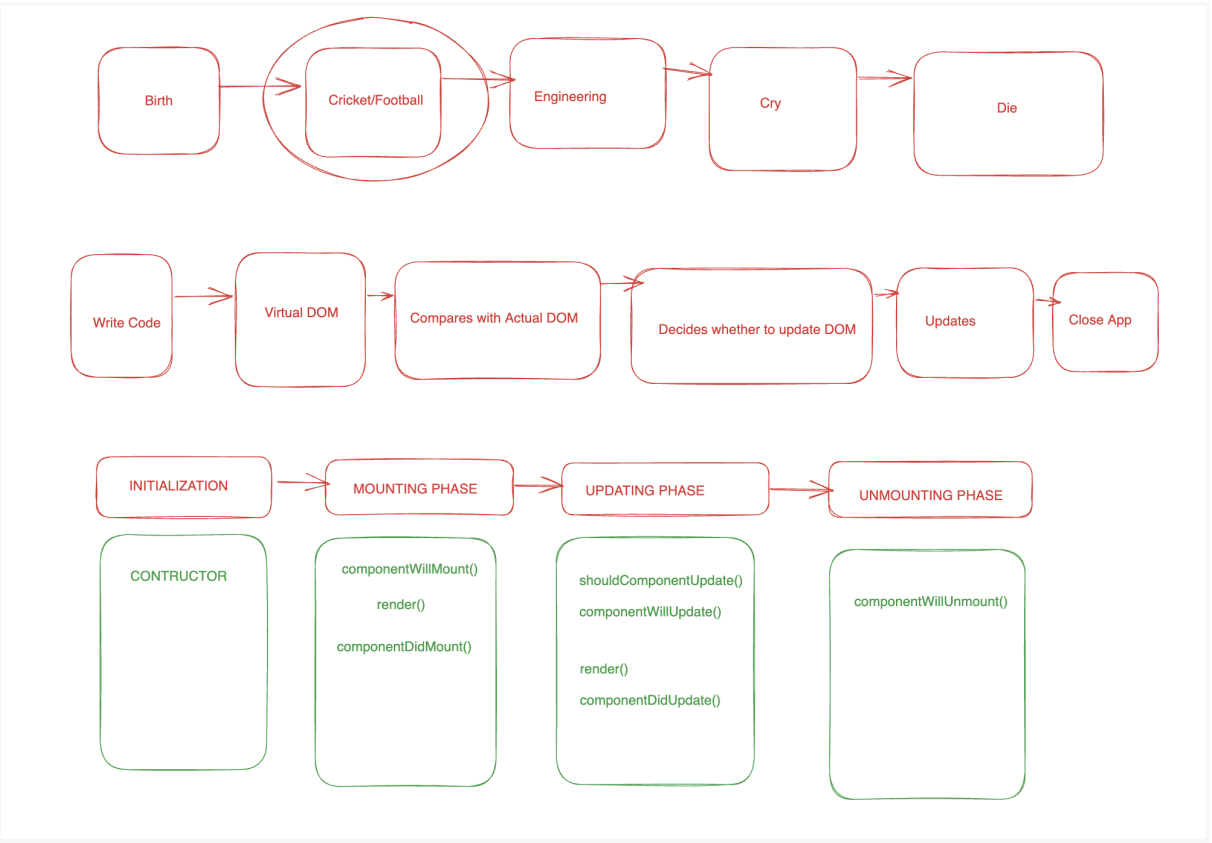
5. componentWillUnmount(): The componentWillUnmount method is called right before the component is unmounted and destroyed. It is used to clean up any resources or subscriptions created in componentDidMount. Here's an example:

```
class MyComponent extends React.Component {
  componentDidMount() {
    this.timerId = setInterval(() => {
      console.log('Tick');
    }, 1000);
  }

  componentWillUnmount() {
    clearInterval(this.timerId);
  }

  render() {
    return <div>Component with timer</div>;
  }
}
```

Combining all the above in a single code we get :

```
class LifecycleExample extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
    console.log('Constructor');
  }

  static getDerivedStateFromProps(props, state) {
    console.log('getDerivedStateFromProps');
    return null;
  }

  componentDidMount() {
    console.log('componentDidMount');
  }

  shouldComponentUpdate(nextProps, nextState) {
```

```
    console.log('shouldComponentUpdate');
    return true;
  }

  getSnapshotBeforeUpdate(prevProps, prevState) {
    console.log('getSnapshotBeforeUpdate');
    return null;
  }

  componentDidUpdate(prevProps, prevState, snapshot) {
    console.log('componentDidUpdate');
  }

  componentWillUnmount() {
    console.log('componentWillUnmount');
  }

  incrementCount = () => {
    this.setState(prevState => ({
      count: prevState.count + 1
    }));
  };

  render() {
    console.log('render');
    return (
      <div>
        <h1>Lifecycle Methods Example</h1>
        <p>Count: {this.state.count}</p>
        <button onClick={this.incrementCount}>Increment</button>
      </div>
    );
  }
}
```



2. Introduction to React Hooks:

React Hooks are functions that allow you to use state and other React features in functional components. Before Hooks were introduced, state and lifecycle management could only be done in class components. Hooks provide a more concise and reusable way to handle state and side effects.

Here's an example of using the `useState` hook to manage state in a functional component:

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

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

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

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

In this example, the `useState` hook is used to create a state variable `count` and a function `setCount` to update its value. Hooks always start with the `use` keyword.

3. What is a React Router?

React Router is a popular routing library for React applications. It allows you to create dynamic single-page applications with multiple views or pages. React Router provides a declarative way to define routes and map them to different components.

Here's an example of using React Router to define routes:

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

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

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

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

```
        </ul>
      </nav>

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

In this example, we use the `BrowserRouter` component from React Router to set up the routing. The `Route` components define the mapping between URLs and components to render. The `Link` components are used for navigation between routes.

4. Conditional Rendering in React:

Conditional rendering in React allows you to show or hide components or elements based on certain conditions. You can use JavaScript expressions or variables to control the rendering behavior.

Here's an example of conditional rendering using the ternary operator:

```
import React from 'react';

function Greeting({ isLoggedIn }) {
 return (
  <div>
   {isLoggedIn ? (
    <h2>Welcome back!</h2>
   ) : (
    <h2>Please log in.</h2

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

In this example, the `Greeting` component renders a different message based on the value of the `isLoggedIn` prop. If `isLoggedIn` is `true`, it displays "Welcome back!", otherwise, it displays "Please log in."

5. Mapping Arrays in JSX:

In JSX, you can use the `map` function to render an array of data into a list of React elements. This is useful when you have an array of data and you want to render each item as a separate component or element.

Here's an example of mapping an array of names to a list of `<li>` elements:

```
import React from 'react';

function NameList({ names }) {
  return (
    <ul>
      {names.map((name, index) => (
        <li key={index}>{name}</li>
      ))}
    </ul>
  );
}
```

In this example, the `map` function is used to iterate over the `names` array and generate a list of `<li>` elements. The `key` prop is necessary to provide a unique identifier for each element.

6. Understanding memo, useMemo, and useCallback:

`memo`, `useMemo`, and `useCallback` are performance optimization hooks in React that help avoid unnecessary re-renders.

- `React.memo`: The `memo` function is a higher-order component (HOC) that memoizes the result of a component rendering. It only re-renders the component if its props have changed.

- `useMemo`: The `useMemo` hook allows you to memoize the result of a function call and cache it until its dependencies change. It is used to optimize expensive calculations or complex data transformations.

- `useCallback`: The `useCallback` hook is similar to `useMemo`, but it memoizes a function instead of a value. It returns a memoized version of the callback function, which only changes if its dependencies change. It is commonly used to optimize callbacks passed to child components.

7. Custom Hooks:

Custom Hooks are reusable hooks that encapsulate common logic and can be shared across multiple components. Custom Hooks allow you to extract component logic into a separate function to promote reusability and code organization.

Here's an example of a custom hook that handles form input state:

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

function useFormInput(initialValue) {
  const [value, setValue] = useState(initialValue);

  function handleChange(e) {
    setValue(e.target.value);
  }

  return {
    value,
    onChange: handleChange,
  };
}
```

In this example, the `useFormInput` hook returns an object with a `value` and `onChange` function. It manages the input state and provides an easy way to handle input changes in multiple components.

ort8. Introduction to Axios and API Calls:

Axios is a popular JavaScript library for making HTTP requests from the browser or Node.js. It provides a simple and powerful API for sending asynchronous requests to APIs and handling the responses.

Here's an example of making an API call using Axios:

```
import axios from 'axios';

axios.get('https://api.example.com/data')
  .then(response => {
    // Handle the response data
    console.log(response.data);
  })
  .catch(error => {
    // Handle errors
    console.error(error);
  });
```

In this example, `axios.get` sends a GET request to the specified URL. The response can be accessed in the `.then` callback, and any errors can be handled in the `.catch` callback.

9. Introduction to Redux:

Redux is a predictable state container for JavaScript applications, commonly used with React. It provides a centralized store to manage the application state and allows you to update the state through actions. Redux follows a unidirectional data flow pattern, which makes it easier to understand and debug complex state changes.

Redux consists of the following key concepts:

- **Store**: The store holds the application state and provides methods to interact with it.

- **Actions**: Actions are plain JavaScript objects that represent an intention to change the state. They are dispatched to the store.

- **Reducers**: Reducers are pure functions that take the current state and an action as inputs and return the new state. They define how the state should change based on the dispatched actions.

- **Dispatch**: Dispatch is a method provided by the store to send actions to the reducers.

- **Selectors**: Selectors are functions that extract specific data from the state.

10. Can React Hooks replace Redux? Context vs. Redux and how to choose one:

React Hooks provide a more lightweight and flexible way to manage state in React components. In some cases, Hooks can be used as an alternative to Redux, especially for smaller applications or simple state management needs. However, for larger applications with complex state requirements or advanced features like time-travel debugging, Redux may still be a better choice.

When deciding between using Context or Redux, consider the following:

- **Context**: Context is a built-in feature of React that allows you to pass data through the component tree without manually passing props. It is suitable for small to medium-sized applications with relatively simple state management needs. Context is easier to set up and has a lower learning curve compared to Redux.

- **Redux**: Redux provides a more powerful and structured solution for managing state, especially for larger applications. It offers features like middleware, time-travel debugging, and a clearly defined data flow. Redux has a steeper learning curve and requires more setup compared to Context.

Prepared By :
Divyansh Singh (LinkedIn : rgndunes)