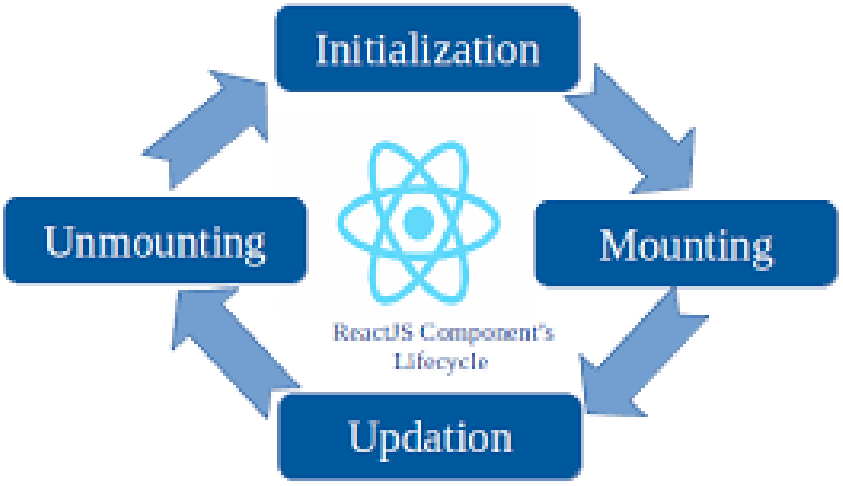
Life Cycle and phases of lifecycle



What Exactly do we mean by lifecycle -

In React, the "life cycle" refers to a series of methods that are called at various stages of a component's existence. Each component goes through a series of phases in its lifecycle, from being initialised and mounted on the page, to being updated with new data, and finally being unmounted and removed from the page. These lifecycle methods allow developers to control the behaviour of components at different stages of their existence and to perform actions like updating data, rendering components, and handling events.

Various components in a lifecycle 👍

1. **Mounting**: In this phase, the component is rendered and inserted into the DOM. This phase has three methods:
- **render():** This method returns a React element that represents the component's UI.

Let's look at an example :

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

- **componentDidMount():** This method is called after the component is mounted and rendered to the DOM. It is often used to fetch data from APIs, set up event listeners, or initialise third-party libraries.

Example -

```
componentDidMount() {
```

```
 setInterval(() => {

  this.setState({ count: this.state.count + 1 });

 }, 1000);

}
```

2. **Updating**: In this phase, the component's state or props are updated, and the component is re-rendered. This phase has five methods:

shouldComponentUpdate(nextProps, nextState): This method is called before the component is updated. It is used to optimise performance by determining if the component needs to be re-rendered or not.

```
shouldComponentUpdate(nextProps, nextState) {

 return nextState.count !== this.state.count;

}
```

- **componentWillUpdate(nextProps, nextState**): This method is called just before the component is updated. It is rarely used and often replaced by the getDerivedStateFromProps() method.

- **render()**: This method is called again to re-render the component with the updated state or props.
- **getSnapshotBeforeUpdate(prevProps, prevState)**: This method is called just before the component is updated and after the render() method. It is used to capture some information from the DOM before it changes. For example:

```
getSnapshotBeforeUpdate(prevProps, prevState) {

 const node = this.refs.list;

 const scrollTop = node.scrollTop;

 const scrollHeight = node.scrollHeight;

 return { scrollTop, scrollHeight };

}
```

componentDidUpdate(prevProps, prevState, snapshot): This method is called after the component is updated and re-rendered. It is often used to update the DOM or third-party libraries. For example:

```
componentDidUpdate(prevProps, prevState, snapshot)
{
```

```
  const node = this.refs.list;

  const { scrollTop, scrollHeight } = snapshot;

  if (scrollHeight > scrollTop) {

    node.scrollTop = scrollHeight - scrollTop;

  }

}
```

3. **Unmounting**: In this phase, the component is removed from the DOM. This phase has one method:

   componentWillUnmount(): This method is called just before the component is unmounted. It is often used to clean up resources, such as event listeners or timers. For example:

```
componentWillUnmount() {

  clearInterval(this.timerID);

}
```

Pure components 👍:

A **pure component** is a type of component that can be used to optimise the performance of a React application. A pure component is a function component that only re-renders when its props have changed.

By default, React re-renders a component whenever its parent component re-renders, regardless of whether the props of the child component have actually changed. This can be a performance issue in large applications, as it can lead to unnecessary re-renders and slow down the application.

Pure components solve this problem by implementing a performance optimization called memoization. Memoization is the process of storing the output of a function and returning the cached result when the same inputs occur again. In the case of a pure component, React stores the output of the component's **render()** function and only re-renders the component when its props have changed.

Let’s consider an example:

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

```
const MyPureComponent = memo(({ text }) => {
  return <div>{text}</div>;
});
```

In the example above, MyPureComponent is a pure component that renders the text prop. It is wrapped with the memo function, which is a higher-order component (HOC) that memoizes the output of the component's render() function based on the equality of its props. This means that if the text prop doesn't change, the component will not re-render.

Note that memo only performs a shallow comparison of the props. This means that if the props contain complex objects or arrays, changes to their nested properties may not be detected. In such cases, it may be necessary to use a custom comparison function with the areEqual parameter of memo.

Using pure components can significantly improve the performance of a React application, especially when dealing with complex components or large datasets. However, it's important to note that pure components are not a silver bullet and should be used judiciously. In some cases, a component may need to re-render even if its props haven't changed, such as when it depends on some external data or has side effects. In such cases, a regular component should be used instead.

Side Effects in react👍

In React, side-effects are actions that a component performs other than rendering, such as fetching data from an API, modifying the DOM, or updating some external state. Side-effects are often necessary for a component to perform its intended functionality, but they can also lead to bugs and performance issues if not managed properly.

In React, pure functions are commonly used to define presentational components, which are components that only take in props and render UI elements based on those props. Presentational components do not perform any side-effects and only rely on their input props to render UI elements.

Here's an example of a pure function component in React:

```
import React from 'react';

const MyComponent = ({ text }) => {
  return <div>{text}</div>;
```

```
}
```

In the example above, **MyComponent** is a **pure function** component that renders the text prop. It doesn't have any side-effects, and only relies on its input props to render UI elements.

Side-effects in React are typically handled using lifecycle methods or hooks. For example, the useEffect hook can be used to perform side-effects in a function component:

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

const MyComponent = () => {
 const [data, setData] = useState([]);

 useEffect(() => {
  fetchData();
 }, []);
```

```
  const fetchData = async () => {
    const response = await
fetch('https://myapi.com/data');
    const data = await response.json();
    setData(data);
  };

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

In the example above, MyComponent uses the useEffect hook to fetch data from an API when the component mounts. The fetched data is stored in the component's state using the useState hook, and rendered as a list of items.

It's important to manage side-effects carefully in React, as they can lead to unpredictable behavior and performance issues. By using pure functions for presentational components and hooks for managing side-effects, you can write maintainable and performant React code.

Use-effect 👍

In React, **useEffect** is a hook that allows you to manage side-effects in function components. Side-effects are actions that a component performs other than rendering, such as fetching data from an API, modifying the DOM, or updating some external state.

The useEffect hook takes two arguments: a callback function that performs the side-effect, and an optional array of dependencies that specify when the side-effect should be performed.

Here's an **example** of using **useEffect** to fetch data from an API:

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


const MyComponent = () => {
```

```
 const [data, setData] = useState([]);


 useEffect(() => {
   const fetchData = async () => {
     const response = await
fetch('https://myapi.com/data');
     const data = await response.json();
     setData(data);
   };
   fetchData();
 }, []);


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

```
};
```

In the example above, MyComponent uses useState to define a state variable data, which is initially an empty array. It then uses useEffect to fetch data from an API when the component mounts. The callback function passed to useEffect is an asynchronous function that fetches the data and updates the state using setData. The **empty dependency array []** ensures that the effect is only run once, when the component mounts.

The useEffect hook can also take an array of dependencies, which specify when the side-effect should be performed. If any of the dependencies change, the effect is run again. For example, to fetch data from an API whenever a prop changes, you could use:

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


const MyComponent = ({ id }) => {
 const [data, setData] = useState({});


 useEffect(() => {
  const fetchData = async () => {
```

```
    const response = await
fetch(`https://myapi.com/data/${id}`);

    const data = await response.json();

    setData(data);

  };

  fetchData();

 }, [id]);


 return (

  <div>

   <h1>{data.name}</h1>

   <p>{data.description}</p>

  </div>

 );

};
```

In the example above, **MyComponent** takes a prop id, which is used to fetch data from an API whenever it changes. The useEffect hook is passed [id] as the dependency array, which ensures that the effect is run whenever id changes.

Using **useEffect** to manage side-effects in React allows you to write declarative code that is easier to reason about and maintain. By specifying when side-effects should be performed, you can avoid **unnecessary re-renders** and ensure that your application remains performant .

SOME INTERVIEW QUESTIONS 👍

1. **What is the React component lifecycle and what are its phases?**

2. **What is the difference between Mounting and Updating phase of the React component lifecycle?**

3. **What lifecycle method is called before a component is mounted?**

4. What is the difference between componentDidMount and componentWillMount methods?

5. What is the purpose of the componentDidUpdate method in React?

6. How do you handle errors in React components using lifecycle methods?

7. What is the componentWillUnmount method used for?

8. How do you prevent unnecessary re-renders of a React component?

9. **What is the difference between shouldComponentUpdate and PureComponent in React?**