# React

## 1. What are the lifecycle methods in React?

React Lifecycle is defined as the series of methods that are auto-invoked in a component's lifecycle.
We have three phases Mounting, Updating, and Unmounting. During mounting, the component is being initialized and added to the DOM. During updating, the component's state or props change, causing a re-render.
During unmounting, the component is removed from the DOM.

## 2. What is the use of constructor?

The constructor is a method used to initialize an object's state in a class.
It is automatically called during the creation of an object in a class

## 3. What are mounting phases?

constructor(props):
* This is the first method called when a component is instantiated. It's used for initializing state, binding event handlers, or any other setup that needs to be done before the component is mounted.

static getDerivedStateFromProps(props, state):
This method is called right before rendering when new props are received. It allows a component to update its state based on changes in props. This is a static method, so it doesn't have access to the this keyword.

render():
This method is required and is responsible for returning the JSX that represents the component's UI. It's called each time the component needs to be re-rendered, which can happen during mounting, updating, or when its parent component re-renders.

componentDidMount():

This method is called after the component has been mounted (i.e., inserted into the DOM). It's commonly used for performing actions that require DOM manipulation, such as fetching data from an API or setting up event listeners.

4. What are updating phases?

static getDerivedStateFromProps(props, state):
This method is called before rendering when new props are received or when the component's state is updated. It allows the component to update its state based on changes in props.

shouldComponentUpdate(nextProps, nextState):
This method is called before rendering when new props or state are received. It allows the component to control whether or not it should re-render. By default, this method returns true, indicating that the component should update. However, you can implement custom logic to optimize performance by preventing unnecessary re-renders.

render():
This method is called to generate the JSX that represents the component's UI. It's called each time the component needs to be re-rendered.

getSnapshotBeforeUpdate(prevProps, prevState):
This method is called right before the changes from the virtual DOM are to be reflected in the actual DOM. It allows the component to capture some information from the DOM before it is potentially changed.

componentDidUpdate(prevProps, prevState, snapshot):
This method is called after the component has been updated and the changes have been reflected in the DOM. It's commonly used for performing actions that need to be taken after the DOM has been updated, such as fetching new data based on the updated props or state.

## 5. What are unmounting phases?

**componentWillUnmount():**

This method is called just before the component is unmounted and destroyed. It's the ideal place to perform any cleanup tasks, such as removing event listeners, clearing timers, or canceling subscriptions, to prevent memory leaks or other unwanted side effects.

## 6. Hooks in react?

Hooks are functions that allow you to use state and other React features in functional components. They were introduced in React 16.8 to address the limitations of classes and provide a more concise and readable way to manage state and side effects in React components.

## 7. useState() Hook ?

Allows functional components to manage state.

Syntax: `const [state, setState] = useState(initialState);`

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

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

## 8. useEffect() Hook?

Allows functional components to perform side effects (such as data fetching, subscriptions, or DOM manipulation) after render.

Syntax: `useEffect(() => { // effect }, [dependencies]);`

Eg:

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

function DataFetcher() {
  const [data, setData] = useState(null);

  useEffect(() => {
    // Fetch data from API
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => {
        setData(data);
      });
  }, []); // Empty dependency array means effect only runs once after the initial render

  return (
    <div>
      {data ? <p>Data: {data}</p> : <p>Loading...</p>}
    </div>
  );
}
```

## 9. useContext() Hook?

Allows functional components to consume context without using Context.Consumer.

Syntax: `const value = useContext(MyContext);`

```jsx
import React, { useContext } from 'react';
import MyContext from './MyContext';

function MyComponent() {
 const value = useContext(MyContext);

 return <p>Context value: {value}</p>;
```

```
}
```

**10. useReducer() Hook:**

Alternative to useState() for managing complex state logic.

Syntax: const [state, dispatch] = useReducer(reducer, initialState);

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

function reducer(state, action) {
 switch (action.type) {
   case 'INCREMENT':
     return { count: state.count + 1 };
   case 'DECREMENT':
     return { count: state.count - 1 };
   default:
     return state;
 }
}

function Counter() {
 const [state, dispatch] = useReducer(reducer, { count: 0 });

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

**11. useMemo() Hook:**

Allows memoizing expensive calculations to optimize performance.

```
Syntax: const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
Example:
 import React, { useMemo } from 'react';

 function MyComponent({ a, b }) {
   const result = useMemo(() => {
     // Expensive calculation
     return a + b;
   }, [a, b]);

   return <p>Result: {result}</p>;
 }
```

## 12. uselayouteffect() Hook?

that is similar to useEffect, but it fires synchronously after all DOM mutations. This means it runs immediately after React has performed all DOM updates. It's useful for scenarios where you need to interact with the DOM immediately after it has been updated

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

function ComponentWithLayoutEffect() {
 const [width, setWidth] = useState(0);
 const ref = useRef(null);

 // Use useLayoutEffect to measure the width of the DOM element after it has been rendered
 useLayoutEffect(() => {
   // Measure the width of the DOM element
   const newWidth = ref.current.offsetWidth;
   // Update state with the measured width
   setWidth(newWidth);
 }, []); // Empty dependency array ensures that the effect runs only once after the initial render
```

```
 return (
   <div>
     <p>Width of the element: {width}px</p>
     <div ref={ref} style={{ width: '200px', height: '100px', backgroundColor:
'lightblue' }}>
       {/* Content */}
     </div>
   </div>
 );
}
```

## 13. Virtual DOM concept?

The Virtual DOM is a concept in React that improves the performance of web applications by minimizing the number of DOM manipulations needed when updating the UI. Here's an explanation in simple terms:

a. What is the DOM?
The Document Object Model (DOM) is a programming interface provided by browsers that represents the structure of an HTML document as a tree of nodes. Each node corresponds to an element, attribute, or text in the document.

b. What is the problem with direct DOM manipulation?
Manipulating the DOM directly can be slow and inefficient, especially when dealing with large and complex web applications. Every time you make a change to the DOM, the browser has to recalculate the layout of the entire page, which can be time-consuming and resource-intensive.

c. What is the Virtual DOM?
The Virtual DOM is a lightweight, in-memory representation of the real DOM. It's a JavaScript object that mirrors the structure of the actual DOM elements in your application.

d. How does React use the Virtual DOM?

*When you render a component in React, it creates a corresponding Virtual DOM representation of that component and its children.

*When the state or props of a component change, React re-renders the component and generates a new Virtual DOM representation.

*React then compares the new Virtual DOM with the previous Virtual DOM to determine the minimal set of DOM operations needed to update the actual DOM.

*Finally, React applies these changes to the real DOM, updating only the parts of the page that have changed.

e. Why is the Virtual DOM efficient?

*Since the Virtual DOM is an in-memory representation, manipulating it is fast and cheap compared to manipulating the actual DOM.

*React optimizes the process of updating the real DOM by batching and minimizing the number of DOM manipulations needed.

*By updating only the parts of the DOM that have changed, React reduces the overall time and resources required for rendering and improves the performance of the application.

## 14. JSX

JavaScript XML allows us to write HTML elements in react and place them, in DOM without any createElement() or appendChild() methods