Core React

1. What is React?

React is an **open-source front-end JavaScript library** that is used for building user interfaces, especially for single-page applications. It is used for handling view layer for web and mobile apps. React was created by Jordan Walke, a software engineer working for Facebook. React was first deployed on Facebook's News Feed in 2011 and on Instagram in 2012.

⬆️ Back to Top

2. What are the major features of React?

The major features of React are:

- It uses **VirtualDOM** instead of RealDOM considering that RealDOM manipulations are expensive.
- Supports **server-side rendering**.
- Follows **Unidirectional** data flow or data binding.
- Uses **reusable/composable** UI components to develop the view.

⬆️ Back to Top

3. What is JSX?

JSX is a XML-like syntax extension to ECMAScript (the acronym stands for *JavaScript XML*). Basically it just provides syntactic sugar for the `React.createElement()` function, giving us expressiveness of JavaScript along with HTML like template syntax.

In the example below text inside `<h1>` tag is returned as JavaScript function to the render function.

```
class App extends React.Component {
  render() {
    return(
      <div>
        <h1>{'Welcome to React world!'}</h1>
      </div>
```

Back to Top

4. What is the difference between Element and Component?

An *Element* is a plain object describing what you want to appear on the screen in terms of the DOM nodes or other components. *Elements* can contain other *Elements* in their props. Creating a React element is cheap. Once an element is created, it is never mutated.

The object representation of React Element would be as follows:

```
const element = React.createElement(
  'div',
  {id: 'login-btn'},
  'Login'
)
```

The above `React.createElement()` function returns an object:

```
{
  type: 'div',
  props: {
    children: 'Login',
    id: 'login-btn'
  }
}
```

And finally it renders to the DOM using `ReactDOM.render()`:

```
<div id='login-btn'>Login</div>
```

Whereas a **component** can be declared in several different ways. It can be a class with a `render()` method or it can be defined as a function. In either case, it takes props as an input, and returns a JSX tree as the output:

```
const Button = ({ onLogin }) =>
  <div id={'login-btn'} onClick={onLogin}>Login</div>
```

Then JSX gets transpiled to a `React.createElement()` function tree:

```
const Button = ({ onLogin }) => React.createElement(
  'div',
  { id: 'login-btn', onClick: onLogin },
  'Login'
)
```

⬆️ Back to Top

5. How to create components in React?

There are two possible ways to create a component.

i. **Function Components:** This is the simplest way to create a component. Those are pure JavaScript functions that accept props object as the first parameter and return React elements:

```
function Greeting({ message }) {
  return <h1>{`Hello, ${message}`}</h1>

}
```

ii. **Class Components:** You can also use ES6 class to define a component. The above function component can be written as:

```
class Greeting extends React.Component {
  render() {
    return <h1>{`Hello, ${this.props.message}`}</h1>
  }
}
```

⬆️ Back to Top

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

If the component needs *state or lifecycle methods* then use class component otherwise use function component. *However, from React 16.8 with the addition of Hooks, you could use state , lifecycle methods and other features that were only available in class component right in your function component.* *So, it is always recommended to use Function components, unless you need a React functionality whose Function component equivalent is not present yet, like Error Boundaries *

⬆️ Back to Top

`React.PureComponent` is exactly the same as `React.Component` except that it handles the `shouldComponentUpdate()` method for you. When props or state changes, *PureComponent* will do a shallow comparison on both props and state. *Component* on the other hand won't compare current props and state to next out of the box. Thus, the component will re-render by default whenever `shouldComponentUpdate` is called.

Back to Top

8. What is state in React?

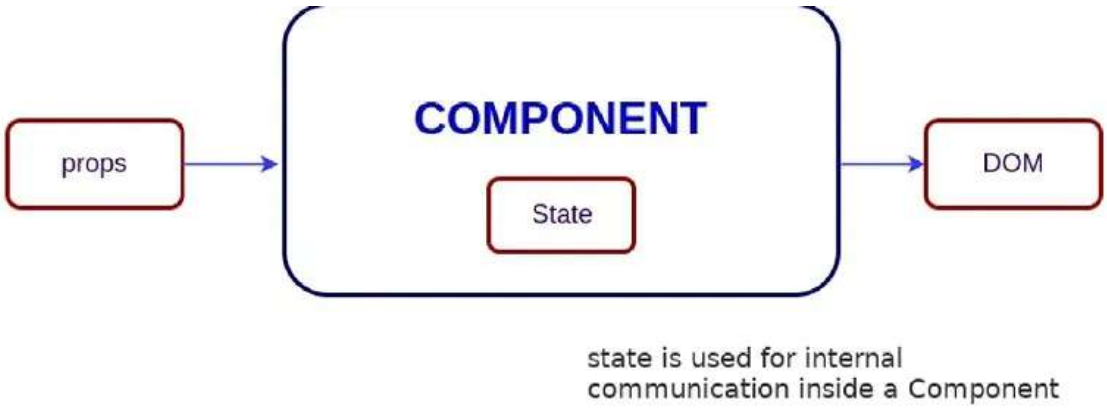
State of a component is an object that holds some information that may change over the lifetime of the component. We should always try to make our state as simple as possible and minimize the number of stateful components.

Let's create a user component with message state,

```
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>
    )
  }
}
```



State is similar to props, but it is private and fully controlled by the component ,i.e., it is not accessible to any other component till the owner component decides to pass it.

Back to Top

9. What are props in React?

Props are inputs to components. They are single values or objects containing a set of values that are passed to components on creation using a naming convention similar to HTML-tag attributes. They are data passed down from a parent component to a child component.

The primary purpose of props in React is to provide following component functionality:

i. Pass custom data to your component.
ii. Trigger state changes.
iii. Use via `this.props.reactProp` inside component's `render()` method.

For example, let us create an element with `reactProp` property:

```
<Element reactProp={'1'} />
```

This `reactProp` (or whatever you came up with) name then becomes a property attached to React's native props object which originally already exists on all components created using React library.

```
props.reactProp
```

Example: Props in Class Based Component

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

class ChildComponent extends React.Component {
    render() {
        return (
            <div>
                <p>{this.props.name}</p>
                <p>{this.props.age}</p>
            </div>
        )
    }
}

class ParentComponent extends React.Component {
    render() {
        return (
            <div>
                <ChildComponent name='John' age='30' />
                <ChildComponent name='Mary' age='25' />
            </div>
        )
    }
}
```

Example: Props in Functional Component

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

const ChildComponent = (props) => {
    return (
        <div>
            <p>{props.name}</p>
            <p>{props.age}</p>
        </div>
    )
}

const ParentComponent = () => {
    return (
        <div>
            <ChildComponent name='John' age='30' />
            <ChildComponent name='Mary' age='25' />
```

Back to Top

10. What is the difference between state and props?

Both *props* and *state* are plain JavaScript objects. While both of them hold information that influences the output of render, they are different in their functionality with respect to component. Props get passed to the component similar to function parameters whereas state is managed within the component similar to variables declared within a function.

Back to Top

11. Why should we not update the state directly?

If you try to update the state directly then it won't re-render the component.

```
//Wrong
this.state.message = 'Hello world'
```

Instead use `setState()` method. It schedules an update to a component's state object. When state changes, the component responds by re-rendering.

```
//Correct
this.setState({ message: 'Hello World' })
```

Note: You can directly assign to the state object either in *constructor* or using latest javascript's class field declaration syntax.

Back to Top

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

The callback function is invoked when setState finished and the component gets rendered. Since `setState()` is **asynchronous** the callback function is used for any post action.

Note: It is recommended to use lifecycle method rather than this callback function.

```
setState({ name: 'John' }, () => console.log('The name has updated and component
```

13. What is the difference between HTML and React event handling?

Below are some of the main differences between HTML and React event handling,

i. In HTML, the event name usually represents in *lowercase* as a convention:

```
<button onclick='activateLasers()'>
```

Whereas in React it follows *camelCase* convention:

```
<button onClick={activateLasers}>
```

ii. In HTML, you can return `false` to prevent default behavior:

```
<a href='#' onclick='console.log("The link was clicked."); return false;' />
```

Whereas in React you must call `preventDefault()` explicitly:

```
function handleClick(event) {
  event.preventDefault()
  console.log('The link was clicked.')
}
```

iii. In HTML, you need to invoke the function by appending `()` Whereas in react you should not append `()` with the function name. (refer "activateLasers" function in the first point for example)

Back to Top

14. How to bind methods or event handlers in JSX callbacks?

There are 3 possible ways to achieve this:

i. **Binding in Constructor:** In JavaScript classes, the methods are not bound by default. The same thing applies for React event handlers defined as class methods. Normally we bind them in constructor.

```
class Foo extends Component {
  constructor(props) {
    super(props);
```

```
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

ii. **Public class fields syntax:** If you don't like to use bind approach then *public class fields syntax* can be used to correctly bind callbacks.

```
handleClick = () => {
  console.log('this is:', this)
}


<button onClick={this.handleClick}>
  {'Click me'}
</button>
```

iii. **Arrow functions in callbacks:** You can use *arrow functions* directly in the callbacks.

```
handleClick() {
    console.log('Click happened');
}
render() {
    return <button onClick={() => this.handleClick()}>Click Me</button>;
}
```

Note: If the callback is passed as prop to child components, those components might do an extra re-rendering. In those cases, it is preferred to go with `.bind()` or *public class fields syntax* approach considering performance.

🔝 Back to Top

15. How to pass a parameter to an event handler or callback?

You can use an *arrow function* to wrap around an *event handler* and pass parameters:

```
<button onClick={() => this.handleClick(id)} />
```

This is an equivalent to calling `.bind` :

```
<button onClick={this.handleClick.bind(this, id)} />
```

Apart from these two approaches, you can also pass arguments to a function which is defined as arrow function

```
<button onClick={this.handleClick(id)} />
handleClick = (id) => () => {
    console.log("Hello, your ticket number is", id)
};
```

Back to Top

16. What are synthetic events in React?

`SyntheticEvent` is a cross-browser wrapper around the browser's native event. Its API is same as the browser's native event, including `stopPropagation()` and `preventDefault()`, except the events work identically across all browsers.

Back to Top

17. What are inline conditional expressions?

You can use either *if statements* or *ternary expressions* which are available from JS to conditionally render expressions. Apart from these approaches, you can also embed any expressions in JSX by wrapping them in curly braces and then followed by JS logical operator `&&`.

```
<h1>Hello!</h1>
{
    messages.length > 0 && !isLogin?
      <h2>
          You have {messages.length} unread messages.
      </h2>
      :
      <h2>
          You don't have unread messages.
      </h2>
}
```

Back to Top

18. What is "key" prop and what is the benefit of using it in arrays of elements?

```
)
```

When you don't have stable IDs for rendered items, you may use the item *index* as a *key* as a last resort:

```
const todoItems = todos.map((todo, index) =>
  <li key={index}>
    {todo.text}
  </li>
)
```

Note:

i. Using *indexes* for *keys* is **not recommended** if the order of items may change. This can negatively impact performance and may cause issues with component state.

ii. If you extract list item as separate component then apply *keys* on list component instead of `li` tag.

iii. There will be a warning message in the console if the `key` prop is not present on list items.

⬆ Back to Top

19. What is the use of refs?

The *ref* is used to return a reference to the element. They *should be avoided* in most cases, however, they can be useful when you need a direct access to the DOM element or an instance of a component.

⬆ Back to Top

20. How to create refs?

There are two approaches

i. This is a recently added approach. *Refs* are created using `React.createRef()` method and attached to React elements via the `ref` attribute. In order to use *refs* throughout the component, just assign the *ref* to the instance property within constructor.

```
class MyComponent extends React.Component {
  constructor(props) {
```

```
    super(props)
    this.myRef = React.createRef()
  }
  render() {
    return <div ref={this.myRef} />
  }
}
```

ii. You can also use ref callbacks approach regardless of React version. For example, the search bar component's input element is accessed as follows,

```
class SearchBar extends Component {
   constructor(props) {
      super(props);
      this.txtSearch = null;
      this.state = { term: '' };
      this.setInputSearchRef = e => {
         this.txtSearch = e;
      }
   }
   onInputChange(event) {
      this.setState({ term: this.txtSearch.value });
   }
   render() {
      return (
         <input
            value={this.state.term}
            onChange={this.onInputChange.bind(this)}
            ref={this.setInputSearchRef} />
      );
   }
}
```

You can also use *refs* in function components using **closures**. **Note**: You can also use inline ref callbacks even though it is not a recommended approach.

Back to Top

21. What are forward refs?

Ref forwarding is a feature that lets some components take a *ref* they receive, and pass it further down to a child.

```
const ButtonElement = React.forwardRef((props, ref) => (
```

```
// Create ref to the DOM button:
const ref = React.createRef();
<ButtonElement ref={ref}>{'Forward Ref'}</ButtonElement>
```

Back to Top

22. Which is preferred option with in callback refs and findDOMNode()?

It is preferred to use *callback refs* over `findDOMNode()` API. Because `findDOMNode()` prevents certain improvements in React in the future.

The **legacy** approach of using `findDOMNode` :

```
class MyComponent extends Component {
  componentDidMount() {
    findDOMNode(this).scrollIntoView()
  }

  render() {
    return <div />
  }
}
```

The recommended approach is:

```
class MyComponent extends Component {
  constructor(props){
    super(props);
    this.node = createRef();
  }
  componentDidMount() {
    this.node.current.scrollIntoView();
  }

  render() {
    return <div ref={this.node} />
  }
}
```

Back to Top

23. Why are String Refs legacy?

If you worked with React before, you might be familiar with an older API where the `ref` attribute is a string, like `ref={'textInput'}`, and the DOM node is accessed as `this.refs.textInput`. We advise against it because *string refs have below issues*, and are considered legacy. String refs were **removed in React v16**.

i. They *force React to keep track of currently executing component*. This is problematic because it makes react module stateful, and thus causes weird errors when react module is duplicated in the bundle.

ii. They are *not composable* — if a library puts a ref on the passed child, the user can't put another ref on it. Callback refs are perfectly composable.

iii. They *don't work with static analysis* like Flow. Flow can't guess the magic that framework does to make the string ref appear on `this.refs`, as well as its type (which could be different). Callback refs are friendlier to static analysis.

iv. It doesn't work as most people would expect with the "render callback" pattern (e.g.)

```
class MyComponent extends Component {
  renderRow = (index) => {
    // This won't work. Ref will get attached to DataTable rather than MyCom
    return <input ref={'input-' + index} />;

    // This would work though! Callback refs are awesome.
    return <input ref={input => this['input-' + index] = input} />;
  }

  render() {
    return <DataTable data={this.props.data} renderRow={this.renderRow} />
  }
}
```

Back to Top

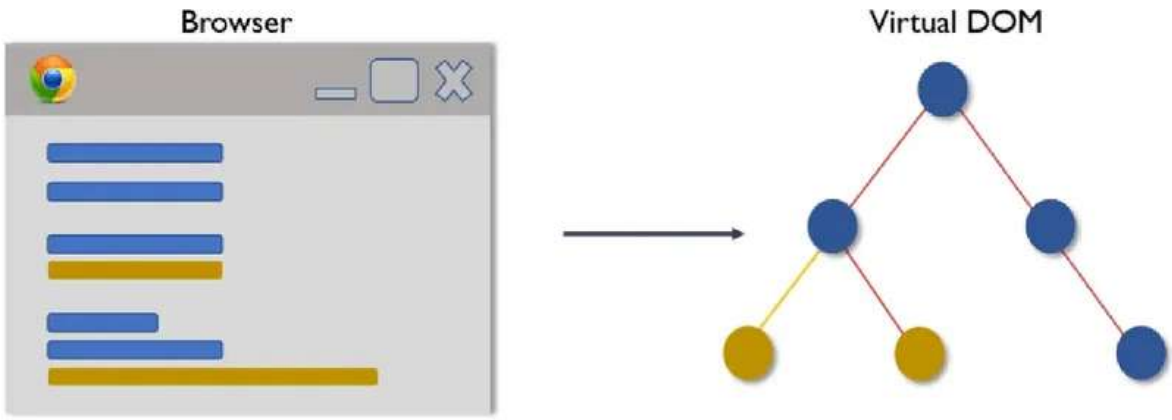
24. What is Virtual DOM?

The *Virtual DOM* (VDOM) is an in-memory representation of *Real DOM*. The representation of a UI is kept in memory and synced with the "real" DOM. It's a step that happens between the render function being called and the displaying of elements on the screen. This entire process is called *reconciliation*.

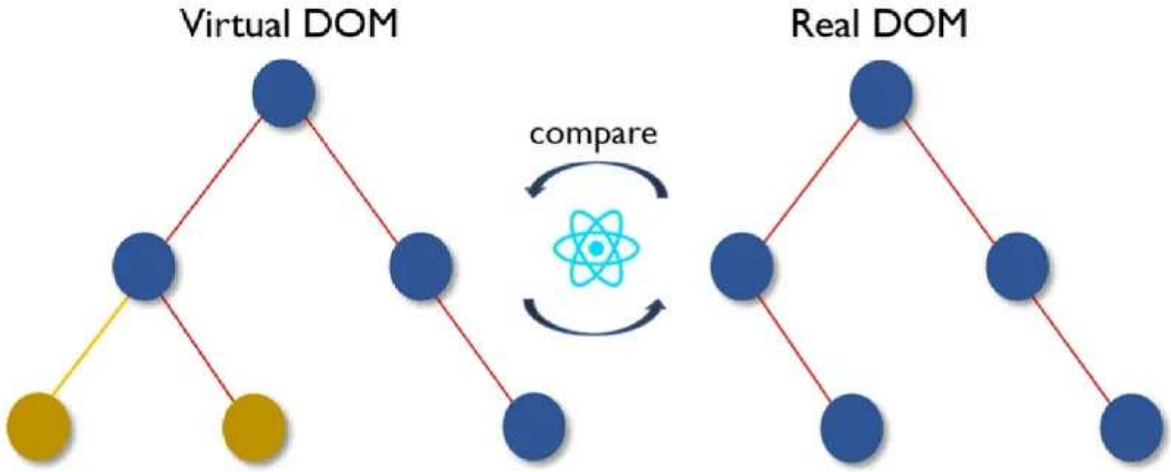
Back to Top

25. How Virtual DOM works?

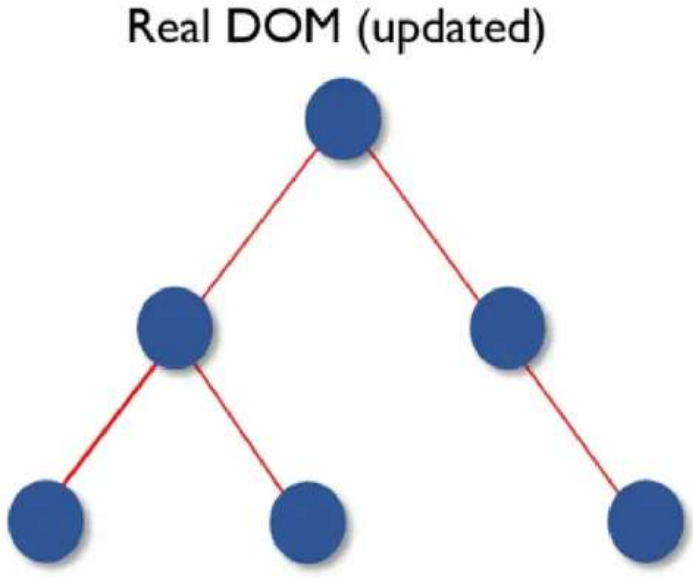
DOM representation.



ii. Then the difference between the previous DOM representation and the new one is calculated.



iii. Once the calculations are done, the real DOM will be updated with only the things that have actually changed.



26. What is the difference between Shadow DOM and Virtual DOM?

The *Shadow DOM* is a browser technology designed primarily for scoping variables and CSS in *web components*. The *Virtual DOM* is a concept implemented by libraries in JavaScript on top of browser APIs.

Back to Top

27. What is React Fiber?

Fiber is the new *reconciliation* engine or reimplementation of core algorithm in React v16. The goal of React Fiber is to increase its suitability for areas like animation, layout, gestures, ability to pause, abort, or reuse work and assign priority to different types of updates; and new concurrency primitives.

Back to Top

28. What is the main goal of React Fiber?

The goal of *React Fiber* is to increase its suitability for areas like animation, layout, and gestures. Its headline feature is **incremental rendering**: the ability to split rendering work into chunks and spread it out over multiple frames.

from documentation

Its main goals are:

i. Ability to split interruptible work in chunks.
ii. Ability to prioritize, rebase and reuse work in progress.
iii. Ability to yield back and forth between parents and children to support layout in React.
iv. Ability to return multiple elements from render().
v. Better support for error boundaries.

Back to Top

29. What are controlled components?

A component that controls the input elements within the forms on subsequent user input is called **Controlled Component**, i.e, every state mutation will have an associated handler function.

For example, to write all the names in uppercase letters, we use handleChange as below,

```
    this.setState({value: event.target.value.toUpperCase()})
  }
```

Back to Top

30. What are uncontrolled components?

The **Uncontrolled Components** are the ones that store their own state internally, and you query the DOM using a ref to find its current value when you need it. This is a bit more like traditional HTML.

In the below UserProfile component, the `name` input is accessed using ref.

```
class UserProfile extends React.Component {
  constructor(props) {
    super(props)
    this.handleSubmit = this.handleSubmit.bind(this)
    this.input = React.createRef()
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.input.current.value)
    event.preventDefault()
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          {'Name:'}
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

In most cases, it's recommend to use controlled components to implement forms. In a controlled component, form data is handled by a React component. The alternative is uncontrolled components, where form data is handled by the DOM itself.

Back to Top

31. What is the difference between createElement and cloneElement?

JSX elements will be transpiled to `React.createElement()` functions to create React elements which are going to be used for the object representation of UI. Whereas `cloneElement` is used to clone an element and pass it new props.

Back to Top

32. What is Lifting State Up in React?

When several components need to share the same changing data then it is recommended to *lift the shared state up* to their closest common ancestor. That means if two child components share the same data from its parent, then move the state to parent instead of maintaining local state in both of the child components.

Back to Top

33. What are the different phases of component lifecycle?

The component lifecycle has three distinct lifecycle phases:

i. **Mounting:** The component is ready to mount in the browser DOM. This phase covers initialization from `constructor()`, `getDerivedStateFromProps()`, `render()`, and `componentDidMount()` lifecycle methods.

ii. **Updating:** In this phase, the component gets updated in two ways, sending the new props and updating the state either from `setState()` or `forceUpdate()`. This phase covers `getDerivedStateFromProps()`, `shouldComponentUpdate()`, `render()`, `getSnapshotBeforeUpdate()` and `componentDidUpdate()` lifecycle methods.

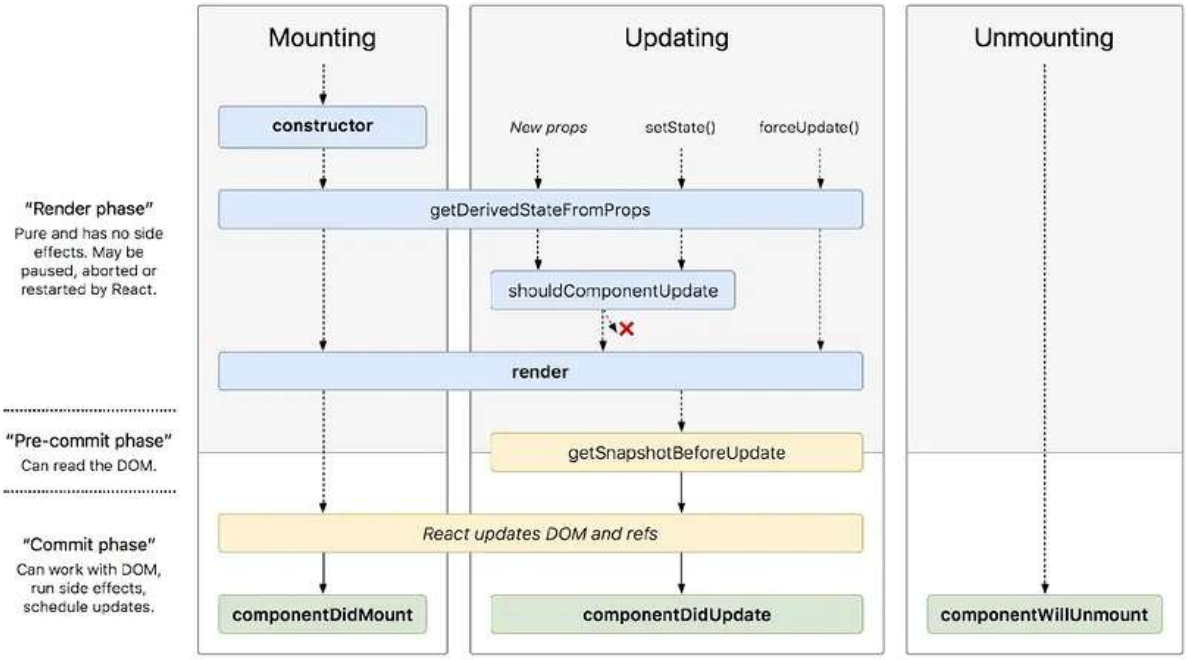
iii. **Unmounting:** In this last phase, the component is not needed and gets unmounted from the browser DOM. This phase includes `componentWillUnmount()` lifecycle method.

It's worth mentioning that React internally has a concept of phases when applying changes to the DOM. They are separated as follows

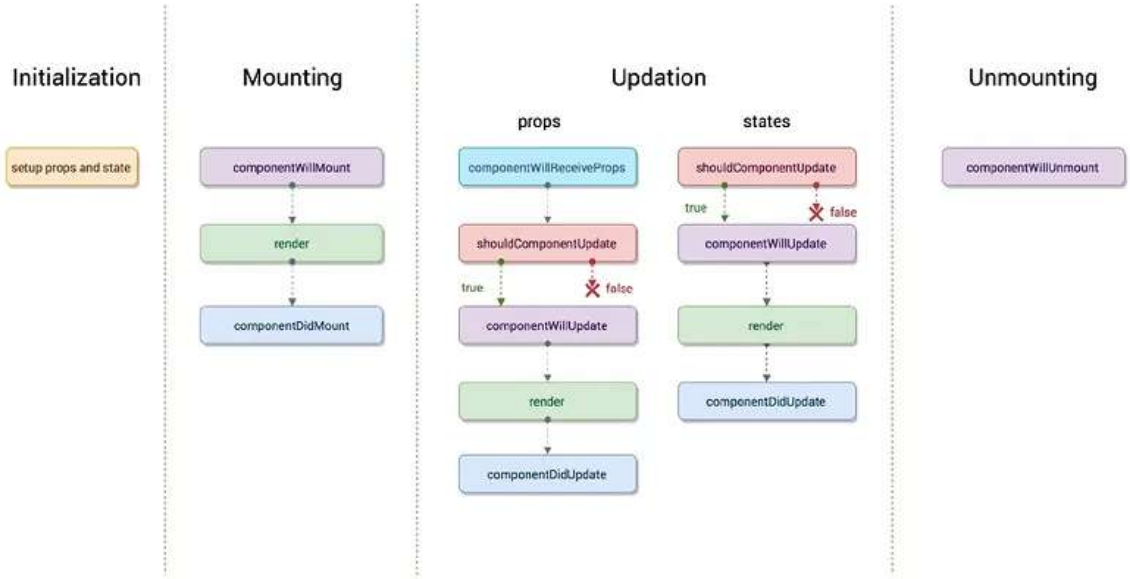
i. **Render** The component will render without any side effects. This applies to Pure components and in this phase, React can pause, abort, or restart the render.

ii. **Pre-commit** Before the component actually applies the changes to the DOM, there is a moment that allows React to read from the DOM through the `getSnapshotBeforeUpdate()`.

iii. **Commit** React works with the DOM and executes the final lifecycles respectively `componentDidMount()` for mounting, `componentDidUpdate()` for updating, and `componentWillUnmount()` for unmounting.



Before React 16.3



Back to Top

34. What are the lifecycle methods of React?

Before React 16.3

- **componentWillMount:** Executed before rendering and is used for App level configuration in your root component.
- **componentDidMount:** Executed after first rendering and here all AJAX requests, DOM or state updates, and set up event listeners should occur.

render after state or props are updated, you can return false value. It is a great place to improve performance as it allows you to prevent a re-render if component receives new prop.

- **componentWillUpdate:** Executed before re-rendering the component when there are props & state changes confirmed by `shouldComponentUpdate()` which returns true.
- **componentDidUpdate:** Mostly it is used to update the DOM in response to prop or state changes.
- **componentWillUnmount:** It will be used to cancel any outgoing network requests, or remove all event listeners associated with the component.

React 16.3+

- **getDerivedStateFromProps:** Invoked right before calling `render()` and is invoked on *every* render. This exists for rare use cases where you need a derived state. Worth reading if you need derived state.
- **componentDidMount:** Executed after first rendering and where all AJAX requests, DOM or state updates, and set up event listeners should occur.
- **shouldComponentUpdate:** Determines if the component will be updated or not. By default, it returns `true`. If you are sure that the component doesn't need to render after the state or props are updated, you can return a false value. It is a great place to improve performance as it allows you to prevent a re-render if component receives a new prop.
- **getSnapshotBeforeUpdate:** Executed right before rendered output is committed to the DOM. Any value returned by this will be passed into `componentDidUpdate()`. This is useful to capture information from the DOM i.e. scroll position.
- **componentDidUpdate:** Mostly it is used to update the DOM in response to prop or state changes. This will not fire if `shouldComponentUpdate()` returns `false`.
- **componentWillUnmount** It will be used to cancel any outgoing network requests, or remove all event listeners associated with the component.

⬆ Back to Top

35. What are Higher-Order Components?

A *higher-order component* (*HOC*) is a function that takes a component and returns a new component. Basically, it's a pattern that is derived from React's compositional nature.

We call them **pure components** because they can accept any dynamically provided child component but they won't modify or copy any behavior from their input