React Notes

Hooks in React : -   React not give allow changes in user interface you can change java script variable but you can't change react user interface through java script variable so here react introduce there hooks concept various hooks concept in react we use there are use - state, use - Memo and may more .

Virtual DOM - Ui Up-dation again and again so you use update final user interface at time use fiber.

React use Fiber Algorithm final user interface updation

1. **What is React?**
React (aka React.js or ReactJS) is an
open-source front-end JavaScript library that is used for building composable user interfaces, especially for single-page applications. It is used for handling view layer for web and mobile apps based on components in a declarative approach.
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 is the history behind React evolution?
The history of ReactJS started in 2010 with the creation of
XHP. XHP is a PHP extension which improved the syntax of the language such that XML document fragments become valid PHP expressions and the primary purpose was used to create custom and reusable HTML elements.
The main principle of this extension was to make front-end code easier to understand and to help avoid cross-site scripting attacks. The project was successful to prevent the malicious content submitted by the scrubbing user.
But there was a different problem with XHP in which dynamic web applications require many roundtrips to the server, and XHP did not solve this problem. Also, the whole UI was re-rendered for small change in the application. Later, the initial prototype of React is created with the name
FaxJ by Jordan inspired from XHP. Finally after sometime React has been introduced as a new library into JavaScript world.

Note: JSX comes from the idea of XHP

⬆ Back to Top
3.
What are the major features of React?
The major features of React are:
- Uses
JSX syntax, a syntax extension of JS that allows developers to write HTML in their JS code.
- It uses
Virtual DOM instead of Real DOM considering that Real DOM manipulations are expensive.
- Supports
server-side rendering which is useful for Search Engine Optimizations(SEO).
- Follows
Unidirectional or one-way data flow or data binding.

◦ Uses
reusable/composable UI components to develop the view.

⬆ Back to Top
4.
What is JSX?

JSX stands for *JavaScript XML* and it is an XML-like syntax extension to ECMAScript. Basically it just provides the syntactic sugar for the `React.createElement(type, props, ...children)` function, giving us expressiveness of JavaScript along with HTML like template syntax.
In the example below, the text inside
`<h1>` tag is returned as JavaScript function to the render function.

```
export default function App() {
  return (
      <h1 className="greeting">{"Hello, this is a JSX Code!"}</h1>
  );
}
```

If you don't use JSX syntax then the respective JavaScript code should be written as below,

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

export default function App() {
  return createElement(
    'h1',
    { className: 'greeting' },
    'Hello, this is a JSX Code!'
  );
}
```
See Class

Note: JSX is stricter than HTML

⬆ Back to Top
5.
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 cannot be mutated.
The JavaScript representation(Without JSX) of React Element would be as follows:

```
const element = React.createElement("div", { id: "login-btn" }, "Login");
```
and this element can be simiplified using JSX

```
  <div id="login-btn">Login</div>
```
The above
`React.createElement()` function returns an object as below:

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

Finally, this element renders to the DOM using
`ReactDOM.render()`.
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 = ({ handleLogin }) => (
  <div id={"login-btn"} onClick={handleLogin}>
    Login
  </div>
);
```

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

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

⬆ Back to Top
6.
How to create components in React?
Components are the building blocks of creating User Interfaces(UI) in React. There are two possible ways to create a component.
  1.
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 to render the output:

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

  2.
Class Components: You can also use ES6 class to define a component. The above function component can be written as a class component:

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

⬆ Back to Top
7.
When to use a Class Component over a Function Component?

After the addition of Hooks(i.e. React 16.8 onwards) it is always recommended to use Function components over Class components in React. Because you could use state, lifecycle methods and other features that were only available in class component present in function component too.
But even there are two reasons to use Class components over Function components.

1. If you need a React functionality whose Function component equivalent is not present yet, like Error Boundaries.
2. In older versions, If the component needs

state or lifecycle methods then you need to use class component.

Note: You can also use reusable react error boundary third-party component without writing any class. i.e, No need to use class components for Error boundaries.
The usage of Error boundaries from the above library is quite straight forward.
Note when using react-error-boundary: ErrorBoundary is a client component. You can only pass props to it that are serializeable or use it in files that have a `"use client";` directive.

```
"use client";

import { ErrorBoundary } from "react-error-boundary";

<ErrorBoundary fallback={<div>Something went wrong</div>}>
  <ExampleApplication />
</ErrorBoundary>
```

🡅 **Back to Top**
8.
What are Pure Components?
Pure components are the components which render the same output for the same state and props. In function components, you can achieve these pure components through memoized
`React.memo()` API wrapping around the component. This API prevents unnecessary re-renders by comparing the previous props and new props using shallow comparison. So it will be helpful for performance optimizations.
But at the same time, it won't compare the previous state with the current state because function component itself prevents the unnecessary rendering by default when you set the same state again.
The syntactic representation of memoized components looks like below,

```
const MemoizedComponent = memo(SomeComponent, arePropsEqual?);
```

Below is the example of how child component(i.e., EmployeeProfile) prevents re-renders for the same props passed by parent component(i.e.,EmployeeRegForm).

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

  const EmployeeProfile = memo(function EmployeeProfile({ name, email }) {
    return (<>
          <p>Name:{name}</p>
          <p>Email: {email}</p>
          </>);
  });
  export default function EmployeeRegForm() {
    const [name, setName] = useState('');
    const [email, setEmail] = useState('');
    return (
      <>
        <label>
          Name: <input value={name} onChange={e => setName(e.target.value)} />
        </label>
        <label>
          Email: <input value={email} onChange={e => setEmail(e.target.value)} />
        </label>
        <hr/>
        <EmployeeProfile name={name}/>
      </>
    );
  }
```

In the above code, the email prop has not been passed to child component. So there won't be any re-renders for email prop change.
In class components, the components extending
`React.PureComponent` instead of `React.Component` become the pure components. When props or state changes, *PureComponent* will do a shallow comparison on both props and state by invoking `shouldComponentUpdate()` lifecycle method.

Note: `React.memo()` is a higher-order component.

⬆ Back to Top
9.
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. The important point is whenever the state object changes, the component re-renders. It is always recommended to make our state as simple as possible and minimize the number of stateful components.

Let's take an example of
User component with message state. Here, **useState** hook has been used to add state to the User component and it returns an array with current state and function to update it.

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

function User() {
  const [message, setMessage] = useState("Welcome to React world");

  return (
    <div>
      <h1>{message}</h1>
    </div>
  );
}
```
See Class
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
10.
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 similar to HTML-tag attributes. Here, the data is passed down from a parent component to a child component.
The primary purpose of props in React is to provide following component functionality:
1. Pass custom data to your component.
2. Trigger state changes.
3. 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) attribute 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`
For example, the usage of props in function component looks like below:

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

The properties from props object can be accessed directly using destructing feature from ES6 (ECMAScript 2015). The above child component can be simplified like below.

```
  const ChildComponent = ({name, age}) => {
      return (
        <div>
          <p>{name}</p>
          <p>{age}</p>
        </div>
      );
    };
```

See Class

⬆ Back to Top
1.
What is the difference between state and props?
In React, both
`state` and `props` are plain JavaScript objects and used to manage the data of a component, but they are used in different ways and have different characteristics. `state` is managed by the component itself and can be updated using the `setState()` function. Unlike props, state can be modified by the component and is used to manage the internal state of the component. Changes in the state trigger a re-render of the component and its children. `props` (short for "properties") are passed to a component by its parent component and are `read-only`, meaning that they cannot be modified by the component itself. props can be used to configure the behavior of a component and to pass data between components.

⬆ Back to Top
2.
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
3.
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 re-rendered")
);
```

⬆ Back to Top
4.
What is the difference between HTML and React event handling?
Below are some of the main differences between HTML and React event handling,
1. In HTML, the event name usually represents in
lowercase as a convention:

```
<button onclick="activateLasers()"></button>
```

Whereas in React it follows
camelCase convention:

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

2. 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.");
}
```

3. 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

5.
How to bind methods or event handlers in JSX callbacks?

There are 3 possible ways to achieve this in class components:

1.
Binding in Constructor: In JavaScript classes, the methods are not bound by default. The same rule applies for React event handlers defined as class methods. Normally we bind them in constructor.

```
class User extends Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log("SingOut triggered");
  }
  render() {
    return <button onClick={this.handleClick}>SingOut</button>;
  }
}
```

2.
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. The Create React App enables this syntax by default.

```
handleClick = () => {
  console.log("SingOut triggered", this);
};
```

```
<button onClick={this.handleClick}>SingOut</button>
```

3.
Arrow functions in callbacks: It is possible to use *arrow functions* directly in the callbacks.

```
handleClick() {
    console.log('SingOut triggered');
}
render() {
    return <button onClick={() => this.handleClick()}>SignOut</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
6.
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
7.
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. The native events can be accessed directly from synthetic events using `nativeEvent` attribute.
Let's take an example of BookStore title search component with the ability to get all native event properties

```
function BookStore() {
  handleTitleChange(e) {
    console.log('The new title is:', e.target.value);
    // 'e' represents synthetic event
    const nativeEvent = e.nativeEvent;
    console.log(nativeEvent);
    e.stopPropogation();
    e.preventDefault();
  }

  return <input name="title" onChange={handleTitleChange} />
}
```

🡅 Back to Top
8.
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

9.

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

A
`key` is a special attribute you **should** include when mapping over arrays to render data. *Key* prop helps React identify which items have changed, are added, or are removed.
Keys should be unique among its siblings. Most often we use ID from our data as
key:

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

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:

1. 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.
2. If you extract list item as separate component then apply
keys on list component instead of `li` tag.
3. There will be a warning message in the console if the
`key` prop is not present on list items.
4. The key attribute accepts either string or number and internally convert it as string type.

🡅 Back to Top

10.

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

11.

How to create refs?

There are two approaches

1. 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} />;
  }
}
```

2. 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
12.
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) => (
  <button ref={ref} className="CustomButton">
    {props.children}
  </button>
));

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

🠉 Back to Top
13.
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**

14.

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**.

1. 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.
2. 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.
3. 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.
4. 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 MyComponent:
    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
15.
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
16.
How Virtual DOM works?
The
Virtual DOM works in three simple steps.

1. Whenever any underlying data changes, the entire UI is re-rendered in Virtual DOM representation.

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

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

🡅 Back to Top
17.
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
18.
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
19.
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:

1. Ability to split interruptible work in chunks.
2. Ability to prioritize, rebase and reuse work in progress.
3. Ability to yield back and forth between parents and children to support layout in React.
4. Ability to return multiple elements from render().
5. Better support for error boundaries.

🠉 Back to Top

20.

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,

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

🠉 Back to Top

21.

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

22.

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

23.
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**
24.
What are the different phases of component lifecycle?
The component lifecycle has three distinct lifecycle phases:

1. **Mounting:** The component is ready to mount in the browser DOM. This phase covers initialization from `constructor()`, `getDerivedStateFromProps()`, `render()`, and `componentDidMount()` lifecycle methods.
2. **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.
3. **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

1. **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.
2. **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()`.
3. **Commit** React works with the DOM and executes the final lifecycles respectively `componentDidMount()` for mounting, `componentDidUpdate()` for updating, and `componentWillUnmount()` for unmounting.

React 16.3+ Phases (or an interactive version)

Before React 16.3

🡅 **Back to Top**
25.
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.
- **componentWillReceiveProps:** Executed when particular prop updates to trigger state transitions.
- **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 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**

26.

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 components.

`const EnhancedComponent = higherOrderComponent(WrappedComponent);`

HOC can be used for many use cases:

1. Code reuse, logic and bootstrap abstraction.
2. Render hijacking.
3. State abstraction and manipulation.
4. Props manipulation.

🠝 **Back to Top**

27.

How to create props proxy for HOC component?

You can add/edit props passed to the component using *props proxy* pattern like this:

```
function HOC(WrappedComponent) {
  return class Test extends Component {
    render() {
      const newProps = {
        title: "New Header",
        footer: false,
        showFeatureX: false,
        showFeatureY: true,
      };

      return <WrappedComponent {...this.props} {...newProps} />;
    }
  };
}
```

⬆ Back to Top
28.
What is context?

Context provides a way to pass data through the component tree without having to pass props down manually at every level.
For example, authenticated users, locale preferences, UI themes need to be accessed in the application by many components.

```
const { Provider, Consumer } = React.createContext(defaultValue);
```

⬆ Back to Top
29.
What is children prop?

Children is a prop (`this.props.children`) that allows you to pass components as data to other components, just like any other prop you use. Component tree put between component's opening and closing tag will be passed to that component as `children` prop.
There are several methods available in the React API to work with this prop. These include `React.Children.map`, `React.Children.forEach`, `React.Children.count`, `React.Children.only`, `React.Children.toArray`.
A simple usage of children prop looks as below,

```
const MyDiv = React.createClass({
  render: function () {
    return <div>{this.props.children}</div>;
  },
});

ReactDOM.render(
  <MyDiv>
    <span>{"Hello"}</span>
    <span>{"World"}</span>
  </MyDiv>,
  node
);
```

⬆ Back to Top
30.
How to write comments in React?

The comments in React/JSX are similar to JavaScript Multiline comments but are wrapped in curly braces.

Single-line comments:

```
<div>
  {/* Single-line comments(In vanilla JavaScript, the single-line comments are represented by double slash(//)) */}
  {`Welcome ${user}, let's play React`}
</div>
```

Multi-line comments:

```
<div>
  {/* Multi-line comments for more than
   one line */}
  {`Welcome ${user}, let's play React`}
</div>
```

⬆ Back to Top

31.

What is the purpose of using super constructor with props argument?

A child class constructor cannot make use of `this` reference until the `super()` method has been called. The same applies to ES6 sub-classes as well. The main reason for passing props parameter to `super()` call is to access `this.props` in your child constructors.

Passing props:

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

    console.log(this.props); // prints { name: 'John', age: 42 }
  }
}
```

Not passing props:

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

    console.log(this.props); // prints undefined

    // but props parameter is still available
    console.log(props); // prints { name: 'John', age: 42 }
  }

  render() {
    // no difference outside constructor
    console.log(this.props); // prints { name: 'John', age: 42 }
  }
}
```

The above code snippets reveals that `this.props` is different only within the constructor. It would be the same outside the constructor.

🡅 Back to Top
32.
What is reconciliation?

`Reconciliation` is the process through which React updates the Browser DOM and makes React work faster. React use a `diffing algorithm` so that component updates are predictable and faster. React would first calculate the difference between the `real DOM` and the copy of DOM `(Virtual DOM)` when there's an update of components. React stores a copy of Browser DOM which is called `Virtual DOM`. When we make changes or add data, React creates a new Virtual DOM and compares it with the previous one. This comparison is done by `Diffing Algorithm`. Now React compares the Virtual DOM with Real DOM. It finds out the changed nodes and updates only the changed nodes in Real DOM leaving the rest nodes as it is. This process is called *Reconciliation*.

🡅 Back to Top
33.
How to set state with a dynamic key name?
If you are using ES6 or the Babel transpiler to transform your JSX code then you can accomplish this with *computed property names*.

```
handleInputChange(event) {
  this.setState({ [event.target.id]: event.target.value })
}
```

🡅 Back to Top
34.
What would be the common mistake of function being called every time the component renders?
You need to make sure that function is not being called while passing the function as a parameter.

```
render() {
  // Wrong: handleClick is called instead of passed as a reference!
  return <button onClick={this.handleClick()}>{'Click Me'}</button>
}
```

Instead, pass the function itself without parenthesis:

```
render() {
  // Correct: handleClick is passed as a reference!
  return <button onClick={this.handleClick}>{'Click Me'}</button>
}
```

🡅 Back to Top
35.
Is lazy function supports named exports?
No, currently
`React.lazy` function supports default exports only. If you would like to import modules which are named exports, you can create an intermediate module that reexports it as the default. It also ensures that tree shaking keeps working and don’t pull unused components. Let's take a component file which exports multiple named components,

```
// MoreComponents.js
export const SomeComponent = /* ... */;
export const UnusedComponent = /* ... */;
```

and reexport
`MoreComponents.js` components in an intermediate file `IntermediateComponent.js`

```
// IntermediateComponent.js
export { SomeComponent as default } from "./MoreComponents.js";
```

Now you can import the module using lazy function as below,

```
import React, { lazy } from "react";
const SomeComponent = lazy(() => import("./IntermediateComponent.js"));
```

⬆ Back to Top
36.
Why React uses `className` over `class` attribute?
The attribute
`class` is a keyword in JavaScript, and JSX is an extension of JavaScript. That's the principle reason why React uses `className` instead of `class`. Pass a string as the `className` prop.

```
render() {
  return <span className={'menu navigation-menu'}>{'Menu'}</span>
}
```

⬆ Back to Top
37.
What are fragments?
It's a common pattern or practice in React for a component to return multiple elements.
Fragments let you group a list of children without adding extra nodes to the DOM. You need to use either or a shorter syntax having empty tag (**<></>**).
Below is the example of how to use fragment inside
Story component.

```
function Story({title, description, date}) {
  return (
      <Fragment>
        <h2>{title}</h2>
        <p>{description}</p>
        <p>{date}</p>
      </Fragment>
    );
}
```

It is also possible to render list of fragments inside a loop with the mandatory
key attribute supplied.

```
function StoryBook() {
  return stories.map(story =>
    <Fragment key={ story.id}>
      <h2>{story.title}</h2>
      <p>{story.description}</p>
      <p>{story.date}</p>
    </Fragment>
    );
}
```

Usually, you don't need to use until unless there is a need of
key attribute. The usage of shorter syntax looks like below.

```
function Story({title, description, date}) {
  return (
      <>
        <h2>{title}</h2>
        <p>{description}</p>
        <p>{date}</p>
      </>
    );
}
```

Back to Top
38.
Why fragments are better than container divs?
Below are the list of reasons to prefer fragments over container DOM elements,

1. Fragments are a bit faster and use less memory by not creating an extra DOM node. This only has a real benefit on very large and deep trees.
2. Some CSS mechanisms like
Flexbox and *CSS Grid* have a special parent-child relationships, and adding divs in the middle makes it hard to keep the desired layout.
3. The DOM Inspector is less cluttered.

Back to Top
39.
What are portals in React?

Portal is a recommended way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

```
ReactDOM.createPortal(child, container);
```

The first argument is any render-able React child, such as an element, string, or fragment. The second argument is a DOM element.

Back to Top
40.
What are stateless components?
If the behaviour of a component is independent of its state then it can be a stateless component. You can use either a function or a class for creating stateless components. But unless you need to use a lifecycle hook in your components, you should go for function components. There are a lot of benefits if you decide to use function components here; they are easy to write, understand, and test, a little faster, and you can avoid the
`this` keyword altogether.

Back to Top
41.
What are stateful components?
If the behaviour of a component is dependent on the
state of the component then it can be termed as stateful component. These *stateful components* are either function

components with hooks or *class components*.
Let's take an example of function stateful component which update the state based on click event,

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

const App = (props) => {
const [count, setCount] = useState(0);
handleIncrement() {
  setCount(count+1);
}

return (
  <>
    <button onClick={handleIncrement}>Increment</button>
    <span>Counter: {count}</span>
  </>
  )
}
```

See Class

⬆ Back to Top

42.

How to apply validation on props in React?

When the application is running in *development mode*, React will automatically check all props that we set on components to make sure they have *correct type*. If the type is incorrect, React will generate warning messages in the console. It's disabled in *production mode* due to performance impact. The mandatory props are defined with `isRequired`.

The set of predefined prop types:

1. `PropTypes.number`
2. `PropTypes.string`
3. `PropTypes.array`
4. `PropTypes.object`
5. `PropTypes.func`
6. `PropTypes.node`
7. `PropTypes.element`
8. `PropTypes.bool`
9. `PropTypes.symbol`
10. `PropTypes.any`

We can define `propTypes` for `User` component as below:

```
import React from "react";
import PropTypes from "prop-types";

class User extends React.Component {
  static propTypes = {
    name: PropTypes.string.isRequired,
    age: PropTypes.number.isRequired,
  };

  render() {
    return (
      <>
        <h1>{`Welcome, ${this.props.name}`}</h1>
        <h2>{`Age, ${this.props.age}`}</h2>
      </>
    );
  }
}
```

Note: In React v15.5 *PropTypes* were moved from `React.PropTypes` to `prop-types` library.

The Equivalent Functional Component

```
import React from "react";
import PropTypes from "prop-types";

function User({ name, age }) {
  return (
    <>
      <h1>{`Welcome, ${name}`}</h1>
      <h2>{`Age, ${age}`}</h2>
    </>
  );
}

User.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number.isRequired,
};
```

⬆ **Back to Top**
43.
What are the advantages of React?
Below are the list of main advantages of React,
1. Increases the application's performance with
Virtual DOM.
2. JSX makes code easy to read and write.
3. It renders both on client and server side (
SSR).
4. Easy to integrate with frameworks (Angular, Backbone) since it is only a view library.
5. Easy to write unit and integration tests with tools such as Jest.

⬆ **Back to Top**
44.
What are the limitations of React?
Apart from the advantages, there are few limitations of React too,
1. React is just a view library, not a full framework.
2. There is a learning curve for beginners who are new to web development.
3. Integrating React into a traditional MVC framework requires some additional configuration.
4. The code complexity increases with inline templating and JSX.
5. Too many smaller components leading to over engineering or boilerplate.

⬆ **Back to Top**
45.
What are error boundaries in React v16?

Error boundaries are components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed.
A class component becomes an error boundary if it defines a new lifecycle method called `componentDidCatch(error, info)` or `static getDerivedStateFromError()` :

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  componentDidCatch(error, info) {
    // You can also log the error to an error reporting service
    logErrorToMyService(error, info);
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>{"Something went wrong."}</h1>;
    }
    return this.props.children;
  }
}
```

After that use it as a regular component:

```
<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>
```

⬆ Back to Top
46.
How are error boundaries handled in React v15?
React v15 provided very basic support for
error boundaries using `unstable_handleError` method. It has been renamed to `componentDidCatch` in React v16.

⬆ Back to Top
47.
What are the recommended ways for static type checking?
Normally we use
PropTypes library (`React.PropTypes` moved to a `prop-types` package since React v15.5) for *type checking* in the React applications. For large code bases, it is recommended to use *static type checkers* such as Flow or TypeScript, that perform type checking at compile time and provide auto-completion features.

⬆ Back to Top
48.
What is the use of `react-dom` package?
The
`react-dom` package provides *DOM-specific methods* that can be used at the top level of your app. Most of the components are not required to use this module. Some of the methods of this package are:

1. `render()`

2. `hydrate()`
3. `unmountComponentAtNode()`
4. `findDOMNode()`
5. `createPortal()`

🡅 Back to Top

49.
What is the purpose of render method of `react-dom`?

This method is used to render a React element into the DOM in the supplied container and return a reference to the component. If the React element was previously rendered into container, it will perform an update on it and only mutate the DOM as necessary to reflect the latest changes.

```
ReactDOM.render(element, container, [callback])
```

If the optional callback is provided, it will be executed after the component is rendered or updated.

🡅 Back to Top

50.
What is ReactDOMServer?

The
`ReactDOMServer` object enables you to render components to static markup (typically used on node server). This object is mainly used for *server-side rendering* (SSR). The following methods can be used in both the server and browser environments:

1. `renderToString()`
2. `renderToStaticMarkup()`

For example, you generally run a Node-based web server like Express, Hapi, or Koa, and you call `renderToString` to render your root component to a string, which you then send as response.

```
// using Express
import { renderToString } from "react-dom/server";
import MyPage from "./MyPage";

app.get("/", (req, res) => {
  res.write(
    "<!DOCTYPE html><html><head><title>My Page</title></head><body>"
  );
  res.write('<div id="content">');
  res.write(renderToString(<MyPage />));
  res.write("</div></body></html>");
  res.end();
});
```

🡅 Back to Top

51.
How to use innerHTML in React?

The
`dangerouslySetInnerHTML` attribute is React's replacement for using `innerHTML` in the browser DOM. Just like `innerHTML`, it is risky to use this attribute considering cross-site scripting (XSS) attacks. You just need to pass a `__html` object as key

and HTML text as value.
In this example MyComponent uses
`dangerouslySetInnerHTML` attribute for setting HTML markup:

```
function createMarkup() {
  return { __html: "First &middot; Second" };
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

🡅 Back to Top
52.
How to use styles in React?
The
`style` attribute accepts a JavaScript object with camelCased properties rather than a CSS string. This is consistent with the DOM style JavaScript property, is more efficient, and prevents XSS security holes.

```
const divStyle = {
  color: "blue",
  backgroundImage: "url(" + imgUrl + ")",
};

function HelloWorldComponent() {
  return <div style={divStyle}>Hello World!</div>;
}
```

Style keys are camelCased in order to be consistent with accessing the properties on DOM nodes in JavaScript (e.g. `node.style.backgroundImage`).

🡅 Back to Top
53.
How events are different in React?
Handling events in React elements has some syntactic differences:
1. React event handlers are named using camelCase, rather than lowercase.
2. With JSX you pass a function as the event handler, rather than a string.

🡅 Back to Top
54.
What will happen if you use `setState()` in constructor?
When you use
`setState()`, then apart from assigning to the object state React also re-renders the component and all its children. You would get error like this: *Can only update a mounted or mounting component.* So we need to use `this.state` to initialize variables inside constructor.

🡅 Back to Top
55.
What is the impact of indexes as keys?
Keys should be stable, predictable, and unique so that React can keep track of elements.
In the below code snippet each element's key will be based on ordering, rather than tied to the data that is being represented. This limits the optimizations that React can do.

```
{
  todos.map((todo, index) => <Todo {...todo} key={index} />);
}
```

If you use element data for unique key, assuming todo.id is unique to this list and stable, React would be able to reorder elements without needing to reevaluate them as much.

```
{
  todos.map((todo) => <Todo {...todo} key={todo.id} />);
}
```

⬆ Back to Top
56.
Is it good to use `setState()` in `componentWillMount()` method?
Yes, it is safe to use
`setState()` inside `componentWillMount()` method. But at the same it is recommended to avoid async initialization in `componentWillMount()` lifecycle method. `componentWillMount()` is invoked immediately before mounting occurs. It is called before `render()`, therefore setting state in this method will not trigger a re-render. Avoid introducing any side-effects or subscriptions in this method. We need to make sure async calls for component initialization happened in `componentDidMount()` instead of `componentWillMount()`.

```
componentDidMount() {
  axios.get(`api/todos`)
    .then((result) => {
      this.setState({
        messages: [...result.data]
      })
    })
}
```

⬆ Back to Top
57.
What will happen if you use props in initial state?
If the props on the component are changed without the component being refreshed, the new prop value will never be displayed because the constructor function will never update the current state of the component. The initialization of state from props only runs when the component is first created.
The below component won't display the updated input value:

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

    this.state = {
      records: [],
      inputValue: this.props.inputValue,
    };
  }

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

Using props inside render method will update the value:

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

    this.state = {
      record: [],
    };
  }

  render() {
    return <div>{this.props.inputValue}</div>;
  }
}
```

⬆ Back to Top
58.
How do you conditionally render components?
In some cases you want to render different components depending on some state. JSX does not render `false` or `undefined`, so you can use conditional *short-circuiting* to render a given part of your component only if a certain condition is true.

```
const MyComponent = ({ name, address }) => (
  <div>
    <h2>{name}</h2>
    {address && <p>{address}</p>}
  </div>
);
```

If you need an
`if-else` condition then use *ternary operator*.

```
const MyComponent = ({ name, address }) => (
  <div>
    <h2>{name}</h2>
    {address ? <p>{address}</p> : <p>{"Address is not available"}</p>}
  </div>
);
```

⬆ Back to Top
59.
Why we need to be careful when spreading props on DOM elements?
When we
spread props we run into the risk of adding unknown HTML attributes, which is a bad practice. Instead we can use prop destructuring with `...rest` operator, so it will add only required props.
For example,

```
const ComponentA = () => (
  <ComponentB isDisplay={true} className={"componentStyle"} />
);

const ComponentB = ({ isDisplay, ...domProps }) => (
  <div {...domProps}>{"ComponentB"}</div>
);
```

⬆ Back to Top
60.

How you use decorators in React?
You can
decorate your *class* components, which is the same as passing the component into a function. **Decorators** are flexible and readable way of modifying component functionality.

```
@setTitle("Profile")
class Profile extends React.Component {
  //....
}

/*
  title is a string that will be set as a document title
  WrappedComponent is what our decorator will receive when
  put directly above a component class as seen in the example above
*/
const setTitle = (title) => (WrappedComponent) => {
  return class extends React.Component {
    componentDidMount() {
      document.title = title;
    }

    render() {
      return <WrappedComponent {...this.props} />;
    }
  };
};
```

Note: Decorators are a feature that didn't make it into ES7, but are currently a *stage 2 proposal*.

⬆ **Back to Top**
61.
How do you memoize a component?
There are memoize libraries available which can be used on function components.
For example
`moize` library can memoize the component in another component.

```
import moize from "moize";
import Component from "./components/Component"; // this module exports a non-memoized component

const MemoizedFoo = moize.react(Component);

const Consumer = () => {
  <div>
    {"I will memoize the following entry:"}
    <MemoizedFoo />
  </div>;
};
```

Update: Since React v16.6.0, we have a `React.memo`. It provides a higher order component which memoizes component unless the props change. To use it, simply wrap the component using React.memo before you use it.

```
const MemoComponent = React.memo(function MemoComponent(props) {
  /* render using props */
});
OR;
export default React.memo(MyFunctionComponent);
```

🡅 **Back to Top**
62.
How you implement Server Side Rendering or SSR?
React is already equipped to handle rendering on Node servers. A special version of the DOM renderer is available, which follows the same pattern as on the client side.

```
import ReactDOMServer from "react-dom/server";
import App from "./App";

ReactDOMServer.renderToString(<App />);
```

This method will output the regular HTML as a string, which can be then placed inside a page body as part of the server response. On the client side, React detects the pre-rendered content and seamlessly picks up where it left off.

🡅 **Back to Top**
63.
How to enable production mode in React?
You should use Webpack's
`DefinePlugin` method to set `NODE_ENV` to `production`, by which it strip out things like propType validation and extra warnings. Apart from this, if you minify the code, for example, Uglify's dead-code elimination to strip out development only code and comments, it will drastically reduce the size of your bundle.

🡅 **Back to Top**
64.
What is CRA and its benefits?
The
`create-react-app` CLI tool allows you to quickly create & run React applications with no configuration step.
Let's create Todo App using
CRA:

```
# Installation
$ npm install -g create-react-app

# Create new project
$ create-react-app todo-app
$ cd todo-app

# Build, test and run
$ npm run build
$ npm run test
$ npm start
```

It includes everything we need to build a React app:
1. React, JSX, ES6, and Flow syntax support.
2. Language extras beyond ES6 like the object spread operator.
3. Autoprefixed CSS, so you don’t need -webkit- or other prefixes.
4. A fast interactive unit test runner with built-in support for coverage reporting.
5. A live development server that warns about common mistakes.
6. A build script to bundle JS, CSS, and images for production, with hashes and sourcemaps.

🡅 **Back to Top**
65.
What is the lifecycle methods order in mounting?
The lifecycle methods are called in the following order when an instance of a component is being created and inserted into the DOM.

1. `constructor()`
2. `static getDerivedStateFromProps()`
3. `render()`
4. `componentDidMount()`

⬆ Back to Top

66.

What are the lifecycle methods going to be deprecated in React v16?

The following lifecycle methods going to be unsafe coding practices and will be more problematic with async rendering.

1. `componentWillMount()`
2. `componentWillReceiveProps()`
3. `componentWillUpdate()`

Starting with React v16.3 these methods are aliased with `UNSAFE_` prefix, and the unprefixed version will be removed in React v17.

⬆ Back to Top

67.

What is the purpose of `getDerivedStateFromProps()` lifecycle method?

The new static `getDerivedStateFromProps()` lifecycle method is invoked after a component is instantiated as well as before it is re-rendered. It can return an object to update state, or `null` to indicate that the new props do not require any state updates.

```
class MyComponent extends React.Component {
  static getDerivedStateFromProps(props, state) {
    // ...
  }
}
```

This lifecycle method along with `componentDidUpdate()` covers all the use cases of `componentWillReceiveProps()`.

⬆ Back to Top

68.

What is the purpose of `getSnapshotBeforeUpdate()` lifecycle method?

The new `getSnapshotBeforeUpdate()` lifecycle method is called right before DOM updates. The return value from this method will be passed as the third parameter to `componentDidUpdate()`.

```
class MyComponent extends React.Component {
  getSnapshotBeforeUpdate(prevProps, prevState) {
    // ...
  }
}
```

This lifecycle method along with `componentDidUpdate()` covers all the use cases of `componentWillUpdate()`.

⬆ Back to Top

69.
Do Hooks replace render props and higher order components?
Both render props and higher-order components render only a single child but in most of the cases Hooks are a simpler way to serve this by reducing nesting in your tree.

🡅 **Back to Top**
70.
What is the recommended way for naming components?
It is recommended to name the component by reference instead of using
`displayName`.
Using
`displayName` for naming component:

```
export default React.createClass({
  displayName: "TodoApp",
  // ...
});
```

The
recommended approach:

```
export default class TodoApp extends React.Component {
  // ...
}
```

also

```
const TodoApp = () => {
  //...
};
export default TodoApp;
```

🡅 **Back to Top**
71.
What is the recommended ordering of methods in component class?

Recommended ordering of methods from *mounting* to *render stage*:
1. `static` methods
2. `constructor()`
3. `getChildContext()`
4. `componentWillMount()`
5. `componentDidMount()`
6. `componentWillReceiveProps()`
7. `shouldComponentUpdate()`
8. `componentWillUpdate()`

9. `componentDidUpdate()`
10. `componentWillUnmount()`
11. click handlers or event handlers like `onClickSubmit()` or `onChangeDescription()`
12. getter methods for render like `getSelectReason()` or `getFooterContent()`
13. optional render methods like `renderNavigation()` or `renderProfilePicture()`
14. `render()`

🠉 **Back to Top**

72.
What is a switching component?
A
switching component is a component that renders one of many components. We need to use object to map prop values to components.
For example, a switching component to display different pages based on
`page` prop:

```
import HomePage from "./HomePage";
import AboutPage from "./AboutPage";
import ServicesPage from "./ServicesPage";
import ContactPage from "./ContactPage";

const PAGES = {
  home: HomePage,
  about: AboutPage,
  services: ServicesPage,
  contact: ContactPage,
};

const Page = (props) => {
  const Handler = PAGES[props.page] || ContactPage;

  return <Handler {...props} />;
};

// The keys of the PAGES object can be used in the prop types to catch dev-time errors.
Page.propTypes = {
  page: PropTypes.oneOf(Object.keys(PAGES)).isRequired,
};
```

🠉 **Back to Top**

73.
Why we need to pass a function to setState()?
The reason behind for this is that
`setState()` is an asynchronous operation. React batches state changes for performance reasons, so the state may not change immediately after `setState()` is called. That means you should not rely on the current state when calling `setState()` since you can't be sure what that state will be. The solution is to pass a function to `setState()`, with the previous state as an argument. By doing this you can avoid issues with the user getting the old state value on access due to the asynchronous nature of `setState()`.
Let's say the initial count value is zero. After three consecutive increment operations, the value is going to be incremented only by one.

```
// assuming this.state.count === 0
this.setState({ count: this.state.count + 1 });
this.setState({ count: this.state.count + 1 });
this.setState({ count: this.state.count + 1 });
// this.state.count === 1, not 3
```

If we pass a function to
`setState()`, the count gets incremented correctly.

```
this.setState((prevState, props) => ({
  count: prevState.count + props.increment,
}));
// this.state.count === 3 as expected
```

(OR)Why function is preferred over object for `setState()`?
React may batch multiple
`setState()` calls into a single update for performance. Because `this.props` and `this.state` may be updated asynchronously, you should not rely on their values for calculating the next state.
This counter example will fail to update as expected:

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

The preferred approach is to call
`setState()` with function rather than object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument.

```
// Correct
this.setState((prevState, props) => ({
  counter: prevState.counter + props.increment,
}));
```

⬆ **Back to Top**
74.
What are React Mixins?

Mixins are a way to totally separate components to have a common functionality. Mixins **should not be used** and can be replaced with *higher-order components* or *decorators*.
One of the most commonly used mixins is
`PureRenderMixin`. You might be using it in some components to prevent unnecessary re-renders when the props and state are shallowly equal to the previous props and state:

```
const PureRenderMixin = require("react-addons-pure-render-mixin");

const Button = React.createClass({
  mixins: [PureRenderMixin],
  // ...
});
```

⬆ **Back to Top**

75.
Why is `isMounted()` an anti-pattern and what is the proper solution?
The primary use case for
`isMounted()` is to avoid calling `setState()` after a component has been unmounted, because it will emit a warning.

```
if (this.isMounted()) {
  this.setState({...})
}
```

Checking
`isMounted()` before calling `setState()` does eliminate the warning, but it also defeats the purpose of the warning.
Using `isMounted()` is a code smell because the only reason you would check is because you think you might be holding a reference after the component has unmounted.
An optimal solution would be to find places where
`setState()` might be called after a component has unmounted, and fix them. Such situations most commonly occur due to callbacks, when a component is waiting for some data and gets unmounted before the data arrives. Ideally, any callbacks should be canceled in `componentWillUnmount()`, prior to unmounting.

⬆ **Back to Top**
76.
What are the Pointer Events supported in React?

Pointer Events provide a unified way of handling all input events. In the old days we had a mouse and respective event listeners to handle them but nowadays we have many devices which don't correlate to having a mouse, like phones with touch surface or pens. We need to remember that these events will only work in browsers that support the *Pointer Events* specification.
The following event types are now available in
React DOM:

1. `onPointerDown`
2. `onPointerMove`
3. `onPointerUp`
4. `onPointerCancel`
5. `onGotPointerCapture`
6. `onLostPointerCapture`
7. `onPointerEnter`
8. `onPointerLeave`
9. `onPointerOver`
10. `onPointerOut`

⬆ **Back to Top**
77.
Why should component names start with capital letter?
If you are rendering your component using JSX, the name of that component has to begin with a capital letter otherwise React will throw an error as an unrecognized tag. This convention is because only HTML elements and SVG tags can begin with a lowercase letter.

```
class SomeComponent extends Component {
  // Code goes here
}
```

You can define component class which name starts with lowercase letter, but when it's imported it should have capital letter. Here lowercase is fine:

```
class myComponent extends Component {
  render() {
    return <div />;
  }
}

export default myComponent;
```

While when imported in another file it should start with capital letter:

```
import MyComponent from "./myComponent";
```
What are the exceptions on React component naming?
The component names should start with an uppercase letter but there are few exceptions to this convention. The lowercase tag names with a dot (property accessors) are still considered as valid component names. For example, the below tag can be compiled to a valid component,

```
     render() {
         return (
           <obj.component/> // `React.createElement(obj.component)`
         )
    }
```

⬆ Back to Top
78.
Are custom DOM attributes supported in React v16?
Yes. In the past, React used to ignore unknown DOM attributes. If you wrote JSX with an attribute that React doesn't recognize, React would just skip it.
For example, let's take a look at the below attribute:

```
<div mycustomattribute={"something"} />
```
Would render an empty div to the DOM with React v15:

```
<div />
```
In React v16 any unknown attributes will end up in the DOM:

```
<div mycustomattribute="something" />
```
This is useful for supplying browser-specific non-standard attributes, trying new DOM APIs, and integrating with opinionated third-party libraries.

⬆ Back to Top
79.
What is the difference between constructor and getInitialState?

You should initialize state in the constructor when using ES6 classes, and `getInitialState()` method when using `React.createClass()`.

Using ES6 classes:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      /* initial state */
    };
  }
}
```

Using `React.createClass()`:

```
const MyComponent = React.createClass({
  getInitialState() {
    return {
      /* initial state */
    };
  },
});
```

Note: `React.createClass()` is deprecated and removed in React v16. Use plain JavaScript classes instead.

⬆ Back to Top

80.

Can you force a component to re-render without calling setState?

By default, when your component's state or props change, your component will re-render. If your `render()` method depends on some other data, you can tell React that the component needs re-rendering by calling `forceUpdate()`.

```
component.forceUpdate(callback);
```

It is recommended to avoid all uses of `forceUpdate()` and only read from `this.props` and `this.state` in `render()`.

⬆ Back to Top

81.

What is the difference between `super()` and `super(props)` in React using ES6 classes?

When you want to access `this.props` in `constructor()` then you should pass props to `super()` method.

Using `super(props)`:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    console.log(this.props); // { name: 'John', ... }
  }
}
```

Using `super()`:

```
class MyComponent extends React.Component {
  constructor(props) {
    super();
    console.log(this.props); // undefined
  }
}
```

Outside
`constructor()` both will display same value for `this.props`.

🡅 Back to Top
82.
How to loop inside JSX?
You can simply use
`Array.prototype.map` with ES6 *arrow function* syntax.
For example, the
`items` array of objects is mapped into an array of components:

```
<tbody>
  {items.map((item) => (
    <SomeComponent key={item.id} name={item.name} />
  ))}
</tbody>
```

But you can't iterate using
`for` loop:

```
<tbody>
  for (let i = 0; i < items.length; i++) {
    <SomeComponent key={items[i].id} name={items[i].name} />
  }
</tbody>
```

This is because JSX tags are transpiled into
function calls, and you can't use statements inside expressions. This may change thanks to `do` expressions which are *stage 1 proposal*.

🡅 Back to Top
83.
How do you access props in attribute quotes?
React (or JSX) doesn't support variable interpolation inside an attribute value. The below representation won't work:

```
<img className="image" src="images/{this.props.image}" />
```

But you can put any JS expression inside curly braces as the entire attribute value. So the below expression works:

```
<img className="image" src={"images/" + this.props.image} />
```

Using
template strings will also work:

```
<img className="image" src={`images/${this.props.image}`} />
```

⬆ Back to Top
84.
What is React proptype array with shape?
If you want to pass an array of objects to a component with a particular shape then use `React.PropTypes.shape()` as an argument to `React.PropTypes.arrayOf()`.

```
ReactComponent.propTypes = {
  arrayWithShape: React.PropTypes.arrayOf(
    React.PropTypes.shape({
      color: React.PropTypes.string.isRequired,
      fontSize: React.PropTypes.number.isRequired,
    })
  ).isRequired,
};
```

⬆ Back to Top
85.
How to conditionally apply class attributes?
You shouldn't use curly braces inside quotes because it is going to be evaluated as a string.

```
<div className="btn-panel {this.props.visible ? 'show' : 'hidden'}">
```

Instead you need to move curly braces outside (don't forget to include spaces between class names):

```
<div className={'btn-panel ' + (this.props.visible ? 'show' : 'hidden')}>
```

Template strings will also work:

```
<div className={`btn-panel ${this.props.visible ? 'show' : 'hidden'}`}>
```

⬆ Back to Top
86.
What is the difference between React and ReactDOM?
The
`react` package contains `React.createElement()`, `React.Component`, `React.Children`, and other helpers related to elements and component classes. You can think of these as the isomorphic or universal helpers that you need to build components. The `react-dom` package contains `ReactDOM.render()`, and in `react-dom/server` we have *server-side rendering* support with `ReactDOMServer.renderToString()` and `ReactDOMServer.renderToStaticMarkup()`.

⬆ Back to Top
87.
Why ReactDOM is separated from React?
The React team worked on extracting all DOM-related features into a separate library called *ReactDOM*. React v0.14 is the first release in which the libraries are split. By looking at some of the packages, `react-native`, `react-art`, `react-canvas`, and `react-three`, it has become clear that the beauty and essence of React has nothing to do with browsers or the DOM.
To build more environments that React can render to, React team planned to split the main React package into two: `react` and `react-dom`. This paves the way to writing components that can be shared between the web version of React and React Native.

🡅 **Back to Top**
88.
How to use React label element?
If you try to render a
`<label>` element bound to a text input using the standard `for` attribute, then it produces HTML missing that attribute and prints a warning to the console.

```
<label for={'user'}>{'User'}</label>
<input type={'text'} id={'user'} />
```

Since
`for` is a reserved keyword in JavaScript, use `htmlFor` instead.

```
<label htmlFor={'user'}>{'User'}</label>
<input type={'text'} id={'user'} />
```

🡅 **Back to Top**
89.
How to combine multiple inline style objects?
You can use
spread operator in regular React:

```
<button style={{ ...styles.panel.button, ...styles.panel.submitButton }}>
  {"Submit"}
</button>
```

If you're using React Native then you can use the array notation:

```
<button style={[styles.panel.button, styles.panel.submitButton]}>
  {"Submit"}
</button>
```

🡅 **Back to Top**
90.
How to re-render the view when the browser is resized?
You can use the
`useState` hook to manage the width and height state variables, and the `useEffect` hook to add and remove the `resize` event listener. The `[]` dependency array passed to useEffect ensures that the effect only runs once (on mount) and not on every re-render.

```
import React, { useState, useEffect } from "react";
function WindowDimensions() {
  const [dimensions, setDimensions] = useState({
    width: window.innerWidth,
    height: window.innerHeight,
  });
  useEffect(() => {
    function handleResize() {
      setDimensions({
        width: window.innerWidth,
        height: window.innerHeight,
      });
    }
    window.addEventListener("resize", handleResize);
    return () => window.removeEventListener("resize", handleResize);
  }, []);

  return (
    <span>
      {dimensions.width} x {dimensions.height}
    </span>
  );
}
```