Q. What is React.js?

React is a JavaScript library created for building fast and interactive user interfaces for web and mobile applications. It is an open source, component-based, front-end library responsible only for the application view layer.

The main objective of ReactJS is to develop User Interfaces (UI) that improves the speed of the apps. It uses virtual DOM (JavaScript object), which improves the performance of the app. The JavaScript virtual DOM is faster than the regular DOM. We can use ReactJS on the client and server-side as well as with other frameworks. It uses component and data patterns that improve readability and helps to maintain larger apps.

Reference:

-

↥

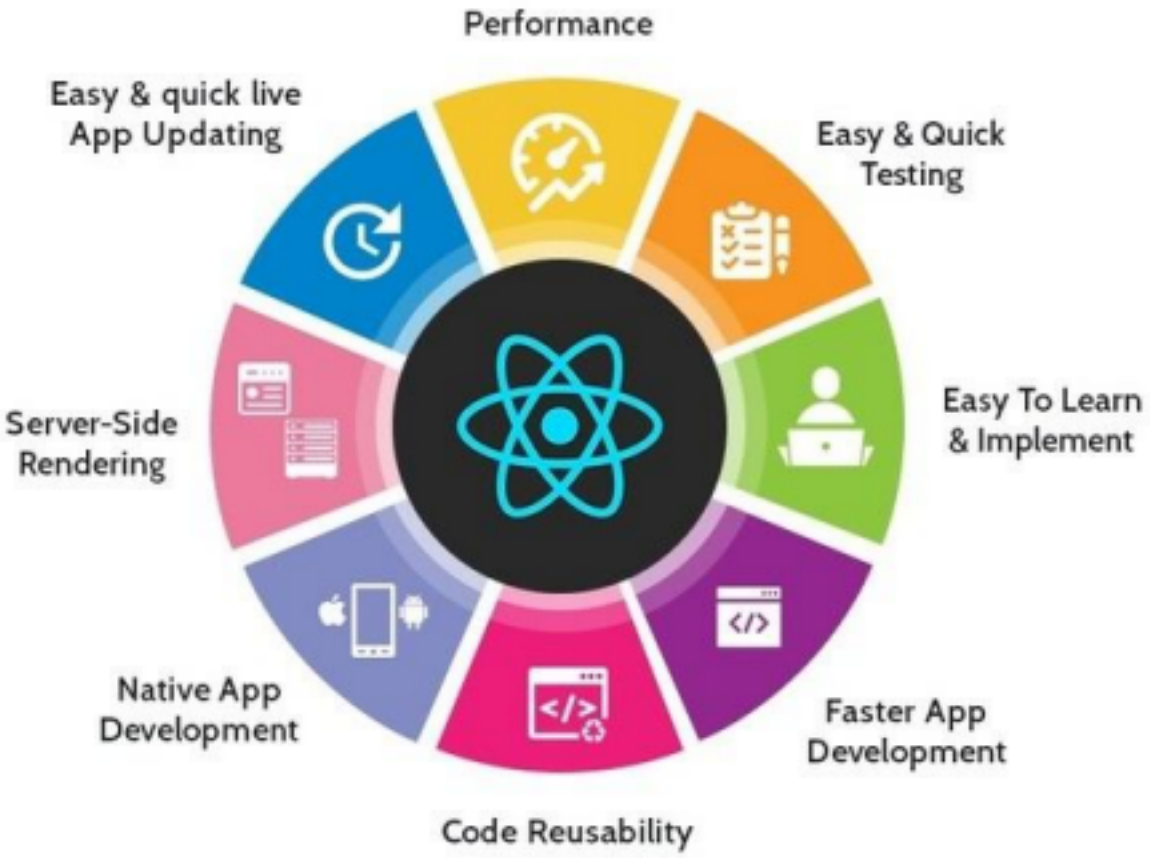
Q. How React works?

React implements a virtual DOM that is basically a DOM tree representation in Javascript. So when it needs to read or write to the DOM, it will use the virtual representation of it. Then the virtual DOM will to find the most efficient way to update the browsers DOM.

Unlike browser DOM elements, React elements are plain objects and are cheap to create. React DOM takes care of updating the DOM to match the React elements. The reason for this is that JavaScript is very fast and it is worth keeping a DOM tree in it to speedup its manipulation.

↥

Q. List some of the major advantages and limitations of React?



Advantages:

- It relies on a virtual-dom to know what is really changing in UI and will re-render only what has really changed, hence better performance wise
- JSX makes components/blocks code readable. It displays how components are plugged or combined with. • React data binding establishes conditions for creation dynamic applications.
- Prompt rendering. Using comprises methods to minimise number of DOM operations helps to optimise updating process and accelerate it. Testable. React native tools are offered for testing, debugging code.
- SEO-friendly. React presents the first-load experience by server side rendering and connecting event-handlers on the side of the user:
    - React.renderComponentToString is called on the server.
    - React.renderComponent() is called on the client side.
    - React preserves markup rendered on the server side, attaches event handlers.

Limitations:

- Learning curve. Being not full-featured framework it is requered in-depth knowledge for integration user interface free library into MVC framework.
- View-orientedness is one of the cons of ReactJS. It should be found 'Model' and 'Controller' to resolve 'View' problem. • Not using isomorphic approach to exploit application leads to search engines indexing problems.

↥

Q. Why does React emphasize on unidirectional data flow?

It is also known as one-way data flow, which means the data has one, and only one way to be transferred to other parts of the application. In essence, this means child components are not able to update the data that is coming from the parent component. In React, data coming from a parent is called **props**.

In React this means that:

- state is passed to the view and to child components
- actions are triggered by the view
- actions can update the state
- the state change is passed to the view and to child components

The view is a result of the application state. State can only change when actions happen. When actions happen, the state is updated. One-way data binding provides us with some key advantages

- Easier to debug, as we know what data is coming from where.
- Less prone to errors, as we have more control over our data.
- More efficient, as the library knows what the boundaries are of each part of the system.

In React, a state is always owned by one component. Any changes made by this state can only affect the components below it, i.e its `children`. Changing state on a component will never affect its parent or its siblings, only the children will be affected. This is the main reason that the state is often moved up in the component tree so that it can be shared between the components that need to access it.

↥

Q. How to declare constant in react?

```
// Constants.js

export const POSTURL = "http://localhost:3000/api/v1/patterns";
export const DELETEURL = "http://localhost:3000/api/v1/patterns/";

export const DeleteButton = require("./images/delete-icon.png");
export const LoadingWheel = require("./images/loading-wheel.gif");

// App.js

import * as Constants from "./Constants";

const employee = {
 emp_id: 10,
 name: "Nakul Agate",
 email: "nakul.agate@email.com"
};

class App extends React.Component {
 render() {
 return (
 <div>
 <div>Employee Details :{JSON.stringify(employee)}</div>
 <div><img src={Constants.LoadingWheel} alt="Loading..." /></div>
 </div>
 );
 }
}
```


↥

Q. What is Destructuring in React?

Destructuring is a convenient way of accessing multiple properties stored in objects and arrays. It was introduced to JavaScript by ES6 and has provided developers with an increased amount of utility when accessing data properties in Objects or Arrays.

When used, destructuring does not modify an object or array but rather copies the desired items from those data structures into variables. These new variables can be accessed later on in a React component.

Example:

```
/**
 * Destructuring in React
 */
import React from "react";

export default function App() {
 // Destructuring
 const [counter, setcounter] = React.useState(0);

 return (
 <>
 <button onClick={() => setcounter(counter + 1)}> Increment </button>
 <button onClick={() => setcounter(counter > 0 ? counter - 1 : 0)}>
 Decrement
 </button>

 <h2>Result: {counter}</h2>
 </>
 );
}
```

☆

↥

Q. Why is it necessary to start component names with a capital letter?

In JSX, lower-case tag names are considered to be HTML tags. However, lower-case tag names with a dot (property accessor) aren't.

When an element type starts with a lowercase letter, it refers to a built-in component like or and results in a string `<div>` or `<span>` passed to `React.createElement`. Types that start with a capital letter like compile to `React.createElement(Foo)` and correspond to a component defined or imported in your JavaScript file.

- `<component />` compiles to `React.createElement('component')` (html tag)
- `<Component />` compiles to `React.createElement(Component)`
- `<obj.component />` compiles to `React.createElement(obj.component)`

↥

Q. What are fragments?

Fragments allows to group a list of children without adding extra nodes to the DOM.

Example:

```
class App extends React.Component {
 render() {
 return (
 <React.Fragment>
 <ChildA />
 <ChildB />
 <ChildC />
```

```
 </React.Fragment>
 )
 }
}
```

Benefits:

- It's a tiny bit faster and has less memory usage (no need to create an extra DOM node). This only has a real benefit on very large and/or deep trees, but application performance often suffers from death by a thousand cuts. This is one cut less.
- Some CSS mechanisms like Flexbox and CSS Grid have a special parent-child relationship, and adding divs in the middle makes it hard to keep the desired layout while extracting logical components.
- The DOM inspector is less cluttered.

↥

Q. What is Virtual DOM?

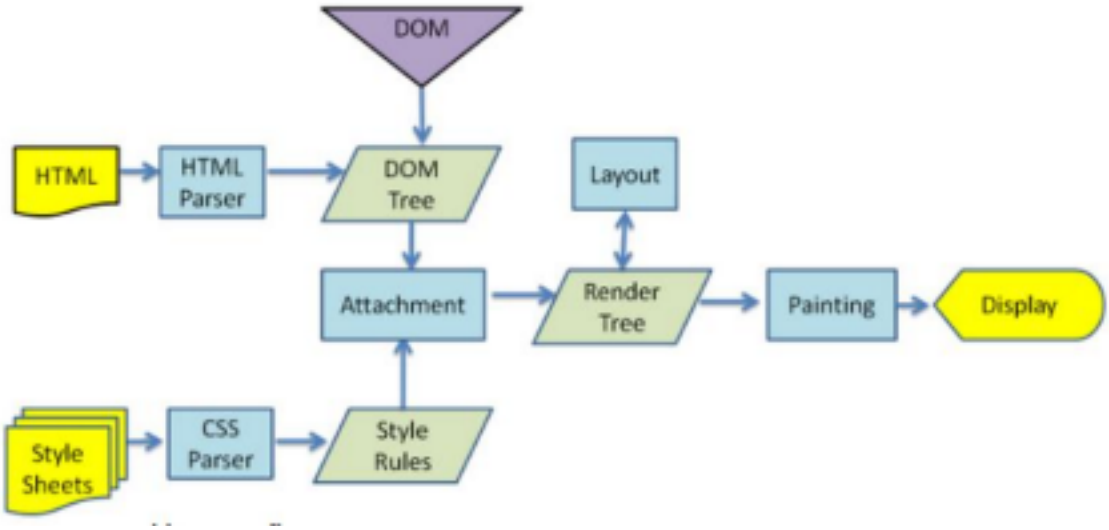
In React, for every **DOM object**, there is a corresponding "virtual DOM object". A virtual DOM object is a representation of a DOM object, like a lightweight copy. A virtual DOM object has the same properties as a real DOM object, but it lacks the real thing's power to directly change what's on the screen.

Manipulating DOM is slow, but manipulating Virtual DOM is fast as nothing gets drawn on the screen. So each time there is a change in the state of our application, virtual DOM gets updated first instead of the real DOM.

☆ **Virtual DOM Example**

↥

Q. What is the difference between ShadowDOM and VirtualDOM?



1. Document Object Model:
It a way of representing a structured document via objects. It is cross-platform and language-independent convention for representing and interacting with data in HTML, XML, and others. Web browsers handle the DOM implementation details, so we can interact with it using JavaScript and CSS.

2. Virtual DOM:

Virtual DOM is any kind of representation of a real DOM. Virtual DOM is about avoiding unnecessary changes to the DOM, which

are expensive performance-wise, because changes to the DOM usually cause re-rendering of the page. It allows to collect several changes to be applied at once, so not every single change causes a re-render, but instead re-rendering only happens once after a set of changes was applied to the DOM.

3. Shadow DOM:

Shadow DOM is mostly about encapsulation of the implementation. A single custom element can implement more-or-less complex logic combined with more-or-less complex DOM. Shadow DOM refers to the ability of the browser to include a subtree of DOM elements into the rendering of a document, but not into the main document DOM tree.

Difference:

The virtual DOM creates an additional DOM. The shadow DOM simply hides implementation details and provides isolated scope for web components.

↥

2. REACT SETUP

Q. How to set up a react project with create react app?

Create React App is an officially supported way to create single-page React applications. It offers a modern build setup with no configuration. This tool is wrapping all of the required dependencies like **Webpack**, **Babel** for React project itself.

Requirements:

The Create React App is maintained by Facebook and can works on any platform, for example, macOS, Windows, Linux, etc. To create a React Project using create-react-app, you need to have installed the following things in your system.

- Node version >= 14
- Visual Studio Code Editor

Installation:

```
npx create-react-app my-app
cd my-app
npm start
```

Output:

Running any of these commands will create a directory called **my-app** inside the current folder. Inside that directory, it will generate the initial project structure and install the transitive dependencies:

```
my-app
├── README.md
├── node_modules
├── package.json
├── .gitignore
├── public
│   ├── favicon.ico
│   ├── index.html
│   ├── logo192.png
│   ├── logo512.png
│   ├── manifest.json
│   └── robots.txt
└── src
    ├── App.css
    ├── App.js
    ├── App.test.js
    ├── index.css
    ├── index.js
    ├── logo.svg
    ├── serviceWorker.js
    └── setupTests.js
```

Reference:

↥

Q. What are the features of create react app?

Create React App is a command-line program that lets us create a new React project easily and build the project into artifacts that we can deploy. It is created by the React team and creates a scaffold to the app.

Below are the list of some of the features provided by create react app.

- React, JSX, ES6, Typescript and Flow syntax support.
- Autoprefixed CSS
- CSS Reset/Normalize
- Live-editing CSS and JS in local development server.
- A fast interactive unit test runner with built-in support for coverage reporting
- A build script to bundle JS, CSS, and images for production, with hashes and sourcemaps
- An offline-first service worker and a web app manifest, meeting all the Progressive Web App criteria. ↥

Q. What does eject do in create react app?

The `create-react-app` commands generate **React App** with an excellent configuration and helps you build your React app with the best practices in mind to optimize it. However, running the `eject` script will remove the single build dependency from your project. That means it will copy the configuration files and the transitive dependencies (e.g. `Webpack`, `Babel`, etc.) as dependencies in the `package.json` file. If you do that, you'll have to ensure that the dependencies are installed before building your project. After running the `eject`, commands like `npm start` and `npm run build` will still work, but they will point to the copied scripts so you can tweak them. It won't be possible to run it again since all scripts will be available except the eject one. ↥

Q. How to put React in production mode?

Create a simple `hello-world-app` using create-react-app.

```
npx create-react-app hello-world-app
```

Modify the `App.js` file as shown below.

```
import './App.css';

function App() {
 return (
 <div className="App">
 <header className="App-header">
 <h1>Hello world app</h1>
 </header>
 </div>
 );
}
export default App;
```

Run the app local server by running the following command

```
npm start
```

On the local server (http://localhost:3000) you can see a simple React app displaying a "hello world" message. The next step is to make this app production-ready for deployment. Inside the root directory run the following command:

```
npm run build
```

This creates a build directory inside the root directory, which bundles your React app and minifies it into simple HTML, CSS, and JavaScript files. This build folder serves your app via a simple en point, `index.html`, where your entire React app resides. Running your app via a remote server means running this `index.html` file on the server.

↥

Q. What are the common folder structures for React?

React doesn't have opinions on how you put files into folders. That said there are a few common approaches popular in the ecosystem you may want to consider.

1. Grouping by features or routes:

One common way to structure projects is to locate CSS, JS, and tests together inside folders grouped by feature or route.

```
common/
 Avatar.js
 Avatar.css
 APIUtils.js
 APIUtils.test.js
feed/
 index.js
 Feed.js
 Feed.css
 FeedStory.js
 FeedStory.test.js
 FeedAPI.js
profile/
 index.js
 Profile.js
 ProfileHeader.js
 ProfileHeader.css
 ProfileAPI.js
```

2. Grouping by file type:

Another popular way to structure projects is to group similar files together, for example:

```
api/
 APIUtils.js
  APIUtils.test.js
 ProfileAPI.js
 UserAPI.js
components/
 Avatar.js
 Avatar.css
 Feed.js
 Feed.css
 FeedStory.js
 FeedStory.test.js
 Profile.js
 ProfileHeader.js
 ProfileHeader.css
```

↥

Q. What are the popular React-specific linter? 1. ESLint:

ESLint is a popular JavaScript linter. There are plugins available that analyse specific code styles. One of the most common for React is an npm package called `eslint-plugin-react`.

```
npm install -g eslint-plugin-react
```

This will install the plugin we need, and in our ESLint config file, we just need a few extra lines.

```
"extends": [
 "eslint:recommended",
 "plugin:react/recommended"
]
```

```
"scripts": {
 "start": "react-scripts start",
 "build": "react-scripts build",
 "test": "react-scripts test --env=jsdom",
 "eject": "react-scripts eject",
```

```
 "lint": "eslint src/**/*.js src/**/*.jsx"
}
```

2. eslint-plugin-jsx-a11y:

It will help fix common issues with accessibility. As JSX offers slightly different syntax to regular HTML, issues with `alt` text and `tabindex`, for example, will not be picked up by regular plugins.

↥

Q. What is the browser support for react applications?

By default, **Create React App** generated project supports all modern browsers. Support for Internet Explorer 9, 10, and 11 requires polyfills. For a set of polyfills to support older browsers, use **react-app-polyfill**.

The `browserslist` configuration controls the outputted JavaScript so that the emitted code will be compatible with the browsers specified.
Example:

```
// package.json

"browserslist": {
 "production": [
 ">0.2%",
 "not dead",
 "not op_mini all"
 ],
 "development": [
 "last 1 chrome version",
 "last 1 firefox version",
 "last 1 safari version"
 ]
}
```

↥

Q. Explain the use of Webpack and Babel in React? 1. Babel:

Babel is a JS transpiler that converts new JS code into old ones. It is a very flexible tool in terms of transpiling. One can easily add presets such as `es2015`, `es2016`, `es2017`, or `env`; so that Babel compiles them to ES5. Babel allows us to have a clean, maintainable code using the latest JS specifications without needing to worry about browser support.
2. Webpack:

Webpack is a modular build tool that has two sets of functionality — Loaders and Plugins. Loaders transform the source code of a module. For example, style-loader adds CSS to DOM using style tags. sass-loader compiles SASS files to CSS. babel-loader transpiles JS code given the presets. Plugins are the core of Webpack. They can do things that loaders can't. For example, there is a plugin called UglifyJS that minifies and uglifies the output of webpack.

3. create-react-app:
create-react-app, a popular tool that lets you set up a React app with just one command. You don't need to get your hands dirty with Webpack or Babel because everything is preconfigured and hidden away from you.

Example: Quick Start

```
npx create-react-app my-app
cd my-app
npm start
```

↥

Q. What is the difference between ReactDOM and React?

The ReactDOM module exposes DOM-specific methods, while React has the core tools intended to be shared by React on different platforms (e.g. React Native).

React package contains: `React.createElement()`, `React.createClass()`, `React.Component()`, `React.PropTypes()`, `React.Children()` **ReactDOM** package contains: `ReactDOM.render()`, `ReactDOM.unmountComponentAtNode()`, `ReactDOM.findDOMNode()`, and react dom/server that including: `ReactDOMServer.renderToString()` and `ReactDOMServer.renderToStaticMarkup()`. **Example:**

```
/**
 * React vs ReactDOM
 */
import { createRoot } from "react-dom/client";

export default function App() {
 return <h1>Hello React</h1>;
}

const rootElement = document.getElementById("root");
const root = createRoot(rootElement);

root.render(<App />);
```

☆

↥

Q. What is ReactDOM?

ReactDOM is a package that provides DOM specific methods that can be used at the top level of a web app to enable an efficient way of managing DOM elements of the web page.

ReactDOM provides the developers with an API containing the following methods

- render()
- findDOMNode()
- unmountComponentAtNode()
- hydrate()
- createPortal()

1. render():

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

Render a React element into the DOM in the supplied container and return a reference to the component (or returns null for stateless components). If the React element was previously rendered into container, this will perform an update on it and only mutate the DOM as necessary to reflect the latest React element. If the optional callback is provided, it will be executed after the component is rendered or updated.

☆

2. hydrate():

```
ReactDOM.hydrate(element, container, callback)
```

This method is equivalent to the `render()` method but is implemented while using server-side rendering. This function attempts to attach event listeners to the existing markup and returns a reference to the component or null if a stateless component was rendered.

☆

3. unmountComponentAtNode():

```
ReactDOM.unmountComponentAtNode(container)
```

This function is used to unmount or remove the React Component that was rendered to a particular container. It returns true if a component was unmounted and false if there was no component to unmount.

☆

4. findDOMNode():

```
ReactDOM.findDOMNode(component)
```

If this component has been mounted into the DOM, this returns the corresponding native browser DOM element. This method is useful for reading values out of the DOM, such as form field values and performing DOM measurements.

☆

5. createPortal():

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

createPortal allow us to render a component into a DOM node that resides outside the current DOM hierarchy of the parent component.

☆

↥

3. REACT JSX

Q. What is JSX?

JSX (**JavaScript Expression**) allows us to write HTML elements in JavaScript and place them in the DOM without any `createElement()` or `appendChild()` methods. JSX converts HTML tags into react elements. React uses JSX for templating instead of regular JavaScript. It is not necessary to use it, however, following are some pros that come with it.

- It is faster because it performs optimization while compiling code to JavaScript.
- It is also type-safe and most of the errors can be caught during compilation.
- It makes it easier and faster to write templates.

When JSX compiled, they actually become regular JavaScript objects. For instance, the code below:

```
const hello = <h1 className = "greet"> Hello World </h1>
```

will be compiled to

```
const hello = React.createElement {
 type: "h1",
 props: {
 className: "greet",
 children: "Hello World"
 }
}
```

Example:

```
export default function App() {
 return (
 <div className="App">
 <h1>Hello World!</h1>
 </div>
 );
}
```

☆

↥

Q. How JSX prevents Injection Attacks?

React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered.

For example, you can embed user input as below,

```
export default class JSXInjectionExample extends React.Component {

 constructor(props) {
 super(props);
 this.state = {
 userContent: `JSX prevents Injection Attacks Example
 <script src="http://example.com/malicious-script.js></script>`
 };
 }

 render() {
 return (
 <div>User content: {this.state.userContent}</div>
 );
 }
}

// Output
User content: JSX prevents Injection Attacks Example
<script src="http://example.com/malicious-script.js></script>
```

☆

↥

Q. What are the benefits of new JSX transform? The React 17 release provides support for a new version of the JSX transform. There are three major benefits of new JSX transform,

- It enables you to use JSX without having to import React.
- The compiled output relatively improves the bundle size.
- The future improvements provides the flexibility to reduce the number of concepts to learn React. ↥

Q. Is it possible to use React without rendering HTML? It is possible with latest version (>=16.2). Below are the possible options:

```
render() {
 return false
}

render() {
 return null
}

render() {
 return []
}

render() {
 return <React.Fragment></React.Fragment>
}

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

Note that React can also run on the server side so, it will be possible to use it in such a way that it doesn't involve any DOM modifications (but maybe only the virtual DOM computation).

↥

Q. How to write comments in React and JSX?

Writing comments in React components can be done just like comment in regular JavaScript classes and functions. **React comments:**

```
function App() {

 // Single line Comment

 /*
 * multi
 * line
 * comment
 **/

 return (
 <h1>My Application</h1>
 );
}
```

JSX comments:

```
export default function App() {
 return (
 <div>
 {/* A JSX comment */}
 <h1>My Application</h1>
 </div>
 );
}
```

↥

Q. How to add custom DOM attributes in JSX?

Custom attributes are supported natively in React 16. This means that adding a custom attribute to an element is now as simple as adding it to a render function, like so:

Example:

```
// 1. Custom DOM Attribute
render() {
 return (
 <div custom-attribute="some-value" />
 );
}

// 2. Data Attribute ( starts with "data-" )
render() {
 return (
 <div data-id="10" />
 );
}

// 3. ARIA Attribute ( starts with "aria-" )
render() {
 return (
 <button aria-label="Close" onClick={onClose} />
 );
}
```

☆

↥

Q. How many outermost elements can be there in a JSX expression?

A JSX expression must have only one outer element. For Example:

```
const headings = (
 <div id = "outermost-element">
 <h1>I am a heading </h1>
 <h2>I am also a heading</h2>
 </div>
)
```

↥

Q. How to loop inside JSX?

You can simply use `Array.prototype.map` with ES6 arrow function syntax.
Example:

```
/**
 * Loop inside JSX
 */
const animals = [
 { id: 1, animal: "Dog" },
 { id: 2, animal: "Bird" },
 { id: 3, animal: "Cat" },
 { id: 4, animal: "Mouse" },
 { id: 5, animal: "Horse" }
];

export default function App() {
 return (
 <ul>
 {animals.map((item) => (
 <li key={item.id}>{item.animal}</li>
 ))}
 </ul>
 );
}
```

☆

↥

Q. How do you print false values in JSX?

In React, boolean values (`true` and `false`), `null`, and `undefined` are valid children, but these values will not be rendered in UI if you put them directly inside {} in JSX.
For example, all these JSX expressions will result in the same empty div:

```
<div />
<div></div>
<div>{false}</div>
<div>{null}</div>
<div>{undefined}</div>
<div>{true}</div>
```

If you want a value like `false`, `true`, `null`, or `undefined` to show in the output, you have to convert it to a string first. `<div>{String(true)}</div>`

```
<div>{String(false)}</div>
<div>{String(undefined)}</div>
<div>{String(null)}</div>
```

In the output, this will render `true`, `false`, `undefined`, and `null` respectively.
☆

↥

Q. How to use React label element?

If you 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'} />
```

↥

Q. How to use InnerHtml in React?

The **innerHTML** is risky because it is easy to expose users to a cross-site scripting (XSS) attack. React provides **dangerouslySetInnerHTML** as a replacement for innerHTML. It allows to set HTML directly from React by using `dangerouslySetInnerHTML` and passing an object with a `__html` key that holds HTML.

Example:

```
function App() {
 return (
 <div
 dangerouslySetInnerHTML={{
 __html: "<h2>This text is set using dangerouslySetInnerHTML</h2>"
 }}
 ></div>
 );
 }
```

↥

Q. How to show and hide elements in React

1. Returning Null:

```
const AddToCart = ({ available }) => {
 if (!available) return null

 return (
 <div className="full tr">
 <button className="product--cart-button">Add to Cart</button>
 </div>
 )
}
```

2. Ternary Display:

When you need to control whether one element vs. another is displayed, or even one element vs. nothing at all (null), you can use the ternary operator embedded inside of a larger portion of JSX.

```
<div className="half">
 <p>{description}</p>

 {remaining === 0 ? (
 <span className="product-sold-out">Sold Out</span>
 ) : (
 <span className="product-remaining">{remaining} remaining</span>
 )}
</div>
```

In this case, if there are no products remaining, we will display "Sold Out"; otherwise we will display the number of products remaining.

3. Shortcut Display:

It involves using a conditional inside of your JSX that looks like `checkIfTrue && <span>display if true</span>`. Because if statements that use `&&` operands stop as soon as they find the first value that evaluates to false, it won't reach the right side (the JSX) if the left side of the equation evaluates to false.

```
<h2>
 <span className="product--title__large">{nameFirst}</span>
 {nameRest.length > 0 && (
 <span className="product--title__small">{nameRest.join(" ")}</span>
 )}
</h2>
```

4. Using Style Property:

```
<div style={{ display: showInfo ? "block" : "none" }}>info</div>
```

↥

4. REACT COMPONENTS

Q. What are React components?

Components are the building blocks of any React app and a typical React app will have many of these. Simply put, a component is a JavaScript class or function that optionally accepts inputs i.e. properties(`props`) and returns a React element that describes how a section of the UI (User Interface) should appear.
In React, a **Stateful Component** is a component that holds some state. A **Stateless component**, by contrast, has no state. Note that both types of components can use props.

1. Stateless Component:

```
import React from 'react'

const ExampleComponent = (props) => {
 return <h1>Stateless Component - {props.message}</h1>;
};

const App = () => {
 const message = 'React Interview Questions'
 return (
 <div>
 <ExampleComponent message={message} />
 </div>
 );
};
export default App;
```

The above example shows a stateless component named ExampleComponent which is inserted in the `<App/>` component. The `ExampleComponent` just comprises of a `<h1>` element. Although the **Stateless component** has no state, it still receives data via props from a parent component.
2. Stateful Component:

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

const ExampleComponent = (props) => {
 const [email, setEmail] = useState(props.defaultEmail)

 const changeEmailHandler = (e) => {
 setEmail(e.target.value)
 }

 return (
 <input type="text" value={email} onChange={changeEmailHandler} />
 );
}


const App = () => {
 const defaultEmail = "suniti.mukhopadhyay@gmail.com"
 return (
 <div>
 <ExampleComponent defaultEmail={defaultEmail} />
 </div>
 );
};

export default App;
```

The above example shows a stateful component named **ExampleComponent** which is inserted in the `<App/>` component. The **ExampleComponent** contains a `<input>`. First of all, in the **ExampleComponent**, we need to assign **defaultEmail** by props to a local **state** by a `useState()` hook in **ExampleComponent**.

Next, we have to pass **email** to **value** property of a input tag and pass a function **changeEmailHandler** to an `onChange()` event for a purpose keeping track of the current value of the input.

↥

Q. What is the difference between Component and Container in React?

The **presentational** components are concerned with the look, **container** components are concerned with making things work. For example, this is a presentational component. It gets data from its props, and just focuses on showing an element

```
/**
 * Presentational Component
 *
 **/
const Users = props => (
 <ul>
 {props.users.map(user => (
 <li>{user}</li>
 ))}
 </ul>
)
```

On the other hand this is a container component. It manages and stores its own data, and uses the presentational component to display it.

```
/**
 * Container Component
 *
 **/
class UsersContainer extends React.Component {
 constructor() {
 this.state = {
 users: []
 }
 }

 componentDidMount() {
 axios.get('/users').then(users =>
 this.setState({ users: users }))
 )
 }

 render() {
 return <Users users={this.state.users} />
 }
}
```

↥

Q. How to import and export components using React.js?

```
// Importing combination
import React, { Component } from 'react';
import ReactDOM from 'react-dom';

// Wrapping components with braces if no default exports
import { Button } from './Button';

// Default exports ( recommended )
import Button from './Button';

class DangerButton extends Component {
 render()
 {
 return <Button color="red" />;
 }
}

export default DangerButton;
```

```
// or export DangerButton;
```

By using default you express that's going to be member in that module which would be imported if no specific member name is provided. You could also express you want to import the specific member called DangerButton by doing so: `import { DangerButton } from './comp/danger-button';` in this case, no default is needed

↥

Q. What is difference between declarative and imperative in React.js?

1. Imperative programming:

It is a programming paradigm that uses statements that change a program's state.

```
const string = "Hi there , I'm a web developer";
let removeSpace = "";
for (let i = 0; i < i.string.length; i++) {
 if (string[i] === " ") removeSpace += "-";
 else removeSpace += string[i];
}
console.log(removeSpace);
```

In this example, we loop through every character in the string, replacing spaces as they occur. Just looking at the code, it doesn't say much. Imperative requires lots of comments in order to understand code. Whereas in the declarative program, the syntax itself describes what should happen and the details of how things happen are abstracted way.

2. Declarative programming:

It is a programming paradigm that expresses the logic of a computation without describing its control flow.
Example:

```
const { render } = ReactDOM
const Welcome = () => (
 <div id="App">
 //your HTML code
 //your react components
 </div>
)
render(
<App />,
document.getElementById('root')
)
```

React is declarative. Here, the **Welcome** component describes the DOM that should be rendered. The render function uses the instructions declared in the component to build the DOM, abstracting away the details of how the DOM is to be rendered. We can clearly see that we want to render our **Welcome** component into the element with the ID of 'target'.

↥

Q. What is the difference between Element and Component?

1. React Element:

It is a simple object that describes a DOM node and its attributes or properties. It is an immutable description object and you can not apply any methods on it.

```
const element = <h1>React Element Example!</h1>;
ReactDOM.render(element, document.getElementById('app'));
```

2. React Component:

It is a function or class that accepts an input and returns a React element. It has to keep references to its DOM nodes and to the instances of the child components.

```
function Message() {
 return <h2>React Component Example!</h2>;
```

```
}
ReactDOM.render(<Message />, document.getElementById('app'));
```

↥

Q. How to conditionally render components in react?

Conditional rendering is a term to describe the ability to render different user interface (UI) markup if a condition is true or false. In React, it allows us to render different elements or components based on a condition.

1. Element Variables:

You can use variables to store elements. This can help you conditionally render a part of the component while the rest of the output doesn't change.

```
function LogInComponent(props) {
 const isLoggedIn = props.isLoggedIn;
 if (isLoggedIn) {
 return <UserComponent />;
 }
 return <GuestComponent />;
}
```

```
ReactDOM.render(
 <LogInComponent isLoggedIn={false} />,
 document.getElementById('root')
);
```

2. Inline If-Else with Conditional Operator:

```
render() {
 const isLoggedIn = this.state.isLoggedIn;
 return (
 <div>
 {isLoggedIn
 ? <LogoutButton onClick={this.handleLogoutClick} />
 : <LoginButton onClick={this.handleLoginClick} />
 }
 </div>
 );
}
```

☆

↥

Q. How to conditionally add attributes to React components?

Inline conditionals in attribute props

```
/**
 * Conditionally add attributes
 */
import React from "react";
```

```
export default function App() {
 const [mood] = React.useState("happy");
```

```
 const greet = () => alert("Hi there! :)");
```

```
 return (
 <button onClick={greet} disabled={"happy" === mood ? false : true}>
 Say Hi
 </button>
 );
}
```

☆

↥

Q. How would you prevent a component from rendering?

React **shouldComponentUpdate()** is a performance optimization method, and it tells React to avoid re-rendering a component, even if state or prop values may have changed. This method only used when a component will stay static or pure.

The React `shouldComponentUpdate()` method return `true` if it needs to re-render or `false` to avoid being re-render. **Syntax:**

```
shouldComponentUpdate(nextProps, nextState){ }
```

Example:

```
/**
 * Prevent a component from rendering
 */
export default class App extends React.Component {
 constructor() {
 super();
 this.state = {
 countOfClicks: 0
 };
 this.handleClick = this.handleClick.bind(this);
 }

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

 shouldComponentUpdate(nextProps, nextState) {
 console.log("this.state.countOfClicks", this.state.countOfClicks);
 console.log("nextState.countOfClicks", nextState.countOfClicks);
 return true;
 }

 render() {
 return (
 <div>
 <h2>shouldComponentUpdate Example</h2>
 <p>Count of clicks: <b>{this.state.countOfClicks}</b></p>
 <button onClick={this.handleClick}>CLICK ME</button>
 </div>
 );
 }
}
```

↥

Q. When would you use StrictMode component in React?

The **StrictMode** is a tool for highlighting potential problems in an application. Like `Fragment`, `StrictMode` does not render any visible UI. It activates additional checks and warnings for its descendants.
Strict mode checks are run in development mode only; they do not impact the production build.

Example:

```
/**
 * StrictMode
 */
import { StrictMode } from "react";
import MyComponent from "./MyComponent";

export default function App() {
 return (
```

```
 <StrictMode>
 <MyComponent />
 </StrictMode>
 );
}
```

React StrictMode, in order to be efficient and avoid potential problems by any side-effects, needs to trigger some methods and lifecycle hooks twice. These are:

- Class component constructor() method
- The render() method
- setState() updater functions (the first argument)
- The static getDerivedStateFromProps() lifecycle
- React.useState() function

Benefits of StrictMode:

- Identifying components with unsafe lifecycles
- Warning about legacy string ref API usage
- Warning about deprecated findDOMNode usage
- Detecting unexpected side effects
- Detecting legacy context API

↥

Q. Why to avoid using setState() after a component has been unmounted?

Calling `setState()` after a component has unmounted will emit a warning. The "setState warning" exists to help you catch bugs, because calling `setState()` on an unmounted component is an indication that your app/component has somehow failed to clean up properly.
Specifically, calling `setState()` in an unmounted component means that your app is still holding a reference to the component after the component has been unmounted - which often indicates a memory leak.
Example:

```
/**
 * setState() in unmounted component
 */
import React, { Component } from "react";
import axios from "axios";

export default class App extends Component {
 _isMounted = false; // flag to check Mounted

 constructor(props) {
 super(props);

 this.state = {
 news: []
 };
 }

 componentDidMount() {
 this._isMounted = true;

 axios
 .get("https://hn.algolia.com/api/v1/search?query=react")
 .then((result) => {
 if (this._isMounted) {
 this.setState({
 news: result.data.hits
 });
 }
```

```
});
}

componentWillUnmount() {
this._isMounted = false;
}

render() {
return (
<ul>
{this.state.news.map((topic) => (
<li key={topic.objectID}>{topic.title}</li>
))}
</ul>
);
}
}
```

Here, even though the component got unmounted and the request resolves eventually, the flag in component will prevent to set the state of the React component after it got unmounted.

☆ ↥

Q. What is Lifting State Up in ReactJS?

The common approach to share **state** between two components is to move the state to common parent of the two components. This approach is called as lifting state up in React.js. With the shared state, changes in state reflect in relevant components simultaneously.

Example:

The App component containing PlayerContent and PlayerDetails component. PlayerContent shows the player name buttons. PlayerDetails shows the details of the in one line.

The app component contains the state for both the component. The selected player is shown once we click on the one of the player button.

```
/**
 * Lifting State Up
 */
import React from "react";
import PlayerContent from "./PlayerContent";
import PlayerDetails from "./PlayerDetails";

export default class App extends React.Component {
 constructor(props) {
 super(props);
 this.state = { selectedPlayer: [0, 0], playerName: "" };
 this.updateSelectedPlayer = this.updateSelectedPlayer.bind(this);
 }
 updateSelectedPlayer(id, name) {
 const arr = [0, 0, 0, 0];
 arr[id] = 1;
 this.setState({
 playerName: name,
 selectedPlayer: arr
 });
 }
 render() {
 return (
 <div>
 <PlayerContent
 active={this.state.selectedPlayer[0]}
 clickHandler={this.updateSelectedPlayer}
 id={0}
 name="Player 1"
 />
 <PlayerContent
 active={this.state.selectedPlayer[1]}
 clickHandler={this.updateSelectedPlayer}
 id={1}
 name="Player 2"
 />
 <PlayerDetails name={this.state.playerName} />
```

```
 </div>
 );
 }
}

/**
 * PlayerContent
 */
import React, { Component } from "react";

export default class PlayerContent extends Component {
 render() {
 return (
 <button
 onClick={() => {
 this.props.clickHandler(this.props.id, this.props.name);
 }}
 style={{ color: this.props.active ? "red" : "blue" }}
 >
 {this.props.name}
 </button>
 );
 }
}

/**
 * PlayerDetails
 */
import React, { Component } from "react";

export default class PlayerDetails extends Component {
 render() {
 return <h2>{this.props.name}</h2>;
 }
}
```

☆

↥

Q. What is "Children" in React?

In React, **children** refer to the generic box whose contents are **unknown** until they're passed from the parent component. Children allows to pass components as data to other components, just like any other prop you use.

The special thing about children is that React provides support through its `ReactElement API` and `JSX`. XML children translate perfectly to React children!
Example:

```
/**
 * Children in React
 */
const Picture = (props) => {
 return (
 <div>
 <img src={props.src}/>
 {props.children}
 </div>
 )
}
```

This component contains an `<img>` that is receiving some props and then it is displaying `{props.children}`. Whenever this component is invoked `{props.children}` will also be displayed and this is just a reference to what is between the opening and closing tags of the component.

```
/**
 * App.js
 */

render () {
 return (
 <div className='container'>
 <Picture key={picture.id} src={picture.src}>
 {/** what is placed here is passed as props.children **/}
 </Picture>
 </div>
 )
```

```
}
```

↥

Q. What is Compound Components in React?

A compound component is a type of component that manages the internal state of a feature while delegating control of the rendering to the place of implementation opposed to the point of declaration. They provide a way to shield feature specific logic from the rest of the app providing a clean and expressive API for consuming the component.

Internally they are built to operate on a set of data that is passed in through children instead of props. Behind the scenes they make use of React's lower level API such as `React.children.map()`, and `React.cloneElement()`. Using these methods, the component is able to express itself in such a way that promotes patterns of composition and extensibility.

Example:

```
function App() {
 return (
 <Menu>
 <MenuButton>
 Actions <span aria-hidden>▾</span>
 </MenuButton>
 <MenuList>
 <MenuItem onSelect={() => alert('Download')}>Download</MenuItem>
 <MenuItem onSelect={() => alert('Copy')}>Create a Copy</MenuItem>
 <MenuItem onSelect={() => alert('Delete')}>Delete</MenuItem>
 </MenuList>
 </Menu>
 )
}
```

In this example, the `<Menu>` establishes some shared implicit state. The `<MenuButton>`, `<MenuList>`, and `<MenuItem>` components each access and/or manipulate that state, and it's all done implicitly. This allows you to have the expressive API you're looking for.

↥

4.1. FUNCTIONAL COMPONENTS

Q. What are functional components in react?

A React functional component is a simple JavaScript function that accepts **props** and returns a React element. It also referred as **stateless** components as it simply accept data and display them in some form.

After the introduction of React Hooks, writing functional components has become the standard way of writing React components in modern applications.

Example:

```
function Welcome(props) {
 return <h1>Hello, {props.name}</h1>;
}

const element = <Welcome name="World!" />;

ReactDOM.render(
 element,
 document.getElementById('root')
);
```

☆

↥

4.2. CLASS COMPONENTS

Q. What are class components in react?

The class component, a stateful/container component, is a regular ES6 class that extends the component class of the React library. It is called a stateful component because it controls how the state changes and the implementation of the component logic. Aside from that, they have access to all the different phases of a React lifecycle method.

Example:

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

const element = <Welcome name="World!" />;

ReactDOM.render(
 element,
 document.getElementById('root')
);
```

☆

↥

Q. What is the recommended ordering of methods in class component?

- static methods
- constructor()
- getChildContext()
- componentWillMount()
- componentDidMount()
- componentWillReceiveProps()
- shouldComponentUpdate()
- componentWillUpdate()
- componentDidUpdate()
- componentWillUnmount()
- click handlers or event handlers like `onClickSubmit()` or `onChangeDescription()`
- getter methods for render like `getSelectReason()` or `getFooterContent()`
- optional render methods like `renderNavigation()` or `renderProfilePicture()`
- render()

↥

Q. How to create a dynamic table in react?

```
/**
 * Generate dynamic table in React
 */
class Table extends React.Component {
 constructor(props) {
```

```
 super(props)
 this.state = {
 employees: [
 { id: 10, name: 'Swarna Sachdeva', email: 'swarna@email.com' },
 { id: 20, name: 'Sarvesh Date', email: 'sarvesh@email.com' },
 { id: 30, name: 'Diksha Meka', email: 'diksha@email.com' }
 ]
 }
 }

 renderTableHeader() {
 let header = Object.keys(this.state.employees[0])
 return header.map((key, index) => {
 return <th key={index}>{key.toUpperCase()}</th>
 })
 }

 renderTableData() {
 return this.state.employees.map((employee, index) => {
 const { id, name, age, email } = employee
 return (
 <tr key={id}>
 <td>{id}</td>
 <td>{name}</td>
 <td>{email}</td>
 </tr>
 )
 })
 }

 render() {
 return (
 <div>
 <h1 id='title'>React Dynamic Table</h1>
 <table id='employees'>
 <tbody>
 <tr>{this.renderTableHeader()}</tr>
 {this.renderTableData()}
 </tbody>
 </table>
 </div>
 )
 }
}
```

☆

↥

Q. How to prevent component from rendering in React?

You can prevent component from rendering by returning `null` based on specific condition. This way it can conditionally render component.
In the example below, the `<WarningBanner />` is rendered depending on the value of the prop called warn. If the value of the prop is `false`, then the component does not render:

```
function WarningBanner(props) {
 if (!props.warn) {
 return null;
 }

 return (
 <div className="warning">
 Warning!
 </div>
 );
}

class Page extends React.Component {
 constructor(props) {
 super(props);
 this.state = {showWarning: true};
 this.handleToggleClick = this.handleToggleClick.bind(this);
 }
```

```
 handleToggleClick() {
 this.setState(state => ({
 showWarning: !state.showWarning
 }));
 }

 render() {
 return (
 <div>
 { /* Prevent component render if value of the prop is false */}
 <WarningBanner warn={this.state.showWarning} />
 <button onClick={this.handleToggleClick}>
 {this.state.showWarning ? 'Hide' : 'Show'}
 </button>
 </div>
 );
 }
}

ReactDOM.render(
 <Page />,
 document.getElementById('root')
);
```

☆

↥

Q. How do you set a timer to update every second?

Using `setInterval()` inside React components allows us to execute a function or some code at specific intervals. A function or block of code that is bound to an interval executes until it is stopped. To stop an interval, we can use the `clearInterval()` method.
Example:

```
class Clock extends React.Component {
 constructor(props) {
 super(props)
 this.state = {
 time: new Date().toLocaleString()
 }
 }
 componentDidMount() {
 this.intervalID = setInterval(
 () => this.tick(),
 1000
 )
 }
 componentWillUnmount() {
 clearInterval(this.intervalID)
 }
 tick() {
 this.setState({
 time: new Date().toLocaleString()
 })
 }
 render() {
 return (
 <p className="App-clock">
 The time is {this.state.time}.
 </p>
 )
 }
}
```

↥

Q. Differentiate between stateful and stateless components?

Stateful and stateless components have many different names. They are also known as:

– Container vs Presentational components
– Smart vs Dumb components

The literal difference is that one has state, and the other does not. That means the stateful components are keeping track of changing data, while stateless components print out what is given to them via props, or they always render the same thing.

Example: Stateful/Container/Smart component

```
class Welcome extends React.Component {
 render() {
 return <h1>This is a React Class Component</h1>;
 }
}
```

Example: Stateless/Presentational/Dumb component

```
function welcome(props) {
 return <h1>This is a React Functional Component</h1>;
}
```

Class Components Functional Components

Functional components are like normal functions which take "props" as the argument and return the required element.

Class components need to extend the component from "React.Component" and create a render function that returns the required element.

They are also known as stateful components. They are also known as stateless components. They implement logic and the

state of the component. They accept some kind of data and display it in the UI. Lifecycle methods can be used inside them.

Lifecycle methods cannot be used inside them. It needs to store state therefore constructors are used. Constructors are not

used in it.

It has to have a "render()" method inside that. It does not require a render method. ↥

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

The `super()` keyword is used to call the parent constructor. `super(props)` would pass `props` to the parent constructor.

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

 // React says we have to define render()
 render() {
 return <div>Hello world</div>
 }
}

export default App
```

Here, `super(props)` would call the `React.Component` constructor passing in props as the argument.

↥

Q. What is the difference between Element,

Component and Component instance in React? 1. React Elements:

A React Element is just a plain old JavaScript Object without own methods. It has essentially four properties:

- **type**: a String representing an HTML tag or a reference referring to a React Component
- **key**: a String to uniquely identify an React Element
- **ref**: a reference to access either the underlying DOM node or React Component Instance)
- **props**: (properties Object)

A React Element is not an instance of a React Component. It is just a simplified "description" of how the React Component Instance to be created should look like.

2. React Components and React Component Instances:

A React Component is used by extending `React.Component`. If a React Component is instantiated it expects a props Object and returns an instance, which is referred to as a React Component Instance.
A React Component can contain state and has access to the React Lifecycle methods. It must have at least a `render` method, which returns a React Element(-tree) when invoked.
Example:

```
/**
 * React Component Instances
 */
import React from 'react'
import ReactDOM from 'react-dom'

class MyComponent extends React.Component {
 constructor(props) {
 super(props)
 console.log('This is a component instance:' + this)
 }

 render() {
 const another_element = <div>Hello, World!</div>
 console.log('This is also an element:' + another_element)
 return another_element
 }
}

console.log('This is a component:' + MyComponent)

const element = <MyComponent/>
console.log('This is an element:' + element)

ReactDOM.render(element, document.getElementById('root'));
```

↥

Q. What does shouldComponentUpdate do and why is it important?

The `shouldComponentUpdate()` method allows Component to exit the Update life cycle if there is no reason to apply a new render. React does not deeply compare `props` by default. When `props` or `state` is updated React assumes we need to re-render the content. The default implementation of this function returns true so to stop the re-render you need to return false here:

```
shouldComponentUpdate(nextProps, nextState) {
 console.log(nextProps, nextState)
 console.log(this.props, this.state)
 return false
}
```

Preventing unnecessary renders

The `shouldComponentUpdate()` method is the first real life cycle optimization method that we can leverage in React. It checks the

current props and state, compares it to the next props and state and then returns true if they are different, or false if they are the same. This method is not called for the initial render or when `forceUpdate()` is used.

↥

Q. What is the purpose of render() function in React?

The React class components uses render() function. It is used to update the UI.

Purpose of render():

- React renders HTML to the web page by using a function called render().
- The purpose of the function is to display the specified HTML code inside the specified HTML element. • In the render() method, we can read props and state and return our JSX code to the root component of our app.
- In the render() method, we cannot change the state, and we cannot cause side effects (such as making an HTTP request to the webserver).

```
/**
 * render() function
 *
 * React v18.0.0
 */
import React from "react";
import { createRoot } from "react-dom/client";

class App extends React.Component {
 render() {
 return <h1>Render() Method Example</h1>;
 }
}

const container = document.getElementById("root");
const root = createRoot(container);
root.render(<App />);
```

☆

↥

4.2.1. REACT LIFECYCLE

Q. What are the different phases of React component lifecycle?

React provides several methods that notify us when certain stage of this process occurs. These methods are called the component lifecycle methods and they are invoked in a predictable order. The lifecycle of the component is divided into four phases.

1. Mounting:

These methods are called in the following order when an instance of a component is being created and inserted into the DOM:

- `constructor()`
- `getDerivedStateFromProps()`
- `render()`
- `componentDidMount()`

2. Updating:

The next phase in the lifecycle is when a component is updated. A component is updated whenever there is a change in

the component's state or props.

React has five built-in methods that gets called, in this order, when a component is updated:

- `getDerivedStateFromProps()`
- `shouldComponentUpdate()`
- `render()`
- `getSnapshotBeforeUpdate()`
- `componentDidUpdate()`

3. Unmounting:

The next phase in the lifecycle is when a component is removed from the DOM, or unmounting as React likes to call it.

- `componentWillUnmount()`

↥

Q. How to make component to perform an action only once when the component initially rendered?

1. Using Class Component:

The `componentDidMount()` lifecycle hook can be used with class components. Any actions defined within a `componentDidMount()` lifecycle hook are called only once when the component is first mounted.

Example:

```
class Homepage extends React.Component {
 componentDidMount() {
 trackPageView('Homepage')
 }
 render() {
 return <div>Homepage</div>
 }
}
```

2. Using Function Component:

The `useEffect()` hook can be used with function components. The `useEffect()` hook is more flexible than the lifecycle methods used for class components. It receives two parameters:

- The first parameter it takes is a callback function to be executed.
- The optional second parameter it takes is an array containing any variables that are to be tracked.

The value passed as the second argument controls when the callback is executed:

- If the second parameter is **undefined**, the callback is executed every time that the component is rendered.
- If the second parameter contains an array of variables, then the callback will be executed as part of the first render cycle and will be executed again each time an item in the array is modified.
- If the second parameter contains an empty array, the callback will be executed only once as part of the first render cycle.

Example:

```
const Homepage = () => {
 useEffect(() => {
 trackPageView('Homepage')
 }, [])

 return <div>Homepage</div>
}
```

Q. What is the typical pattern for rendering a list of components from an array of data?

The usual pattern for rendering lists of components often ends with delegating all of the responsibilities of each child component to the entire list container component. But with a few optimizations, we can make a change in a child component not cause the parent component to re-render.

Example: using custom `shouldComponentUpdate()`

```
/**
 * shouldComponentUpdate()
 */
class AnimalTable extends React.Component<Props, never> {
 shouldComponentUpdate(nextProps: Props) {
 return !nextProps.animalIds.equals(this.props.animalIds);
 }
 ...
```

Here, `shouldComponentUpdate()` will return false if the props its receiving are equal to the props it already has. And because the AnimalTable is receiving just a List of string IDs, a change in the adoption status won't cause AnimalTable to receive a different set of IDs.

Q. What is difference between useEffect() vs componentDidMount()?

In react when we use class based components we get access to lifecycle methods (like `componentDidMount()`, `componentDidUpdate(), etc). But when we want use a functional component and also we want to use lifecycle methods, then using useEffect() we can implement those lifecycle methods.

1. componentDidMount():

The `componentDidMount()` and `useEffect()` run after the mount. However useEffect() runs after the paint has been committed to the screen as opposed to before. This means we would get a flicker if needed to read from the DOM, then synchronously set state to make new UI.

The `useLayoutEffect()` was designed to have the same timing as componentDidMount(). So `useLayoutEffect(fn, [])` is a much closer match to componentDidMount() than useEffect(fn, []) -- at least from a timing standpoint.

```
/**
 * componentDidMount() in Class Component
 */
import React, { Component } from "react";

export default class SampleComponent extends Component {
 componentDidMount() {
 // code to run on component mount
 }
 render() {
 return <>componentDidMount Example</>;
 }
}
```

2. useEffect():

```
/**
 * useEffect() in Functional Component
 */
import React, { useEffect } from "react";

const SampleComponent = () => {
 useEffect(() => {
 // code to run on component mount
 }, []);
 return <>useEffect Example</>;
};
export default SampleComponent;
```

When `useEffect()` is used to get data from server.

- The first argument is a callback that will be fired after browser layout and paint. Therefore it does not block the painting process of the browser.
- The second argument is an array of values (usually props).
- If any of the value in the array changes, the callback will be fired after every render.
- When it is not present, the callback will always be fired after every render.
- When it is an empty list, the callback will only be fired once, similar to componentDidMount.

↥

Q. Why is a component constructor called only once?

React's **reconciliation algorithm** assumes that without any information to the contrary, if a custom component appears in the same place on subsequent renders, it's the same component as before, so reuses the previous instance rather than creating a new one.

If you give each component a unique key `prop`, React can use the key change to infer that the component has actually been substituted and will create a new one from scratch, giving it the full component lifecycle.

```
renderContent() {
 if (this.state.activeItem === 'item-one') {
 return (
 <Content title="First" key="first" />
 )
 } else {
 return (
 <Content title="Second" key="second" />
 )
 }
}
```

↥

Q. What is difference between componentDidMount() and componentWillMount()?

componentDidMount()

The `componentDidMount()` is executed after the first render only on the client side. This is where AJAX requests and DOM or state updates should occur. This method is also used for integration with other JavaScript frameworks and any functions with delayed execution such as `setTimeout()` or `setInterval()`.

Example:

```
import React, { Component } from 'react'

class App extends Component {

 constructor(props) {
 super(props)
 this.state = {
 data: 'Alex Belfort'
 }
 }

 getData(){
 setTimeout(() => {
 console.log('Our data is fetched')
 this.setState({
 data: 'Hello Alex'
 })
 }, 1000)
 }

 componentDidMount() {
 this.getData()
 }
```

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

export default App
```

componentWillMount()

The `componentWillMount()` method is executed before rendering, on both the server and the client side. `componentWillMount()` method is the least used lifecycle method and called before any HTML element is rendered. It is useful when we want to do something programatically right before the component mounts.

Example:

```
import React, { Component } from 'react'

class App extends Component {

 constructor(props) {
 super(props)
 this.state = {
 data: 'Alex Belfort'
 }
 }
 componentWillMount() {
 console.log('First this called')
 }

 getData() {
 setTimeout(() => {
 console.log('Our data is fetched')
 this.setState({
 data: 'Hello Alex'
 })
 }, 1000)
 }

 componentDidMount() {
 this.getData()
 }

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

export default App
```

Q. Is it good to use setState() in componentWillMount() method?

Avoid async initialization in `componentWillMount()`.
`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. Make async calls for component initialization in `componentDidMount()` instead of `componentWillMount() function componentDidMount() {`

```
 axios.get(`api/messages`)
 .then((result) => {
 const messages = result.data
 console.log("COMPONENT WILL Mount messages : ", messages);
 this.setState({
 messages: [...messages.content]
 })
 })
}
```

Q. How to use componentWillUnmount() with Functional Components in React?

The **useEffect()** can be used to manage API calls, as well as implementing **componentWillMount()**, and **componentWillUnmount()**.

If we pass an empty array as the second argument, it tells useEffect to fire on component load. This is the only time it will fire.

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

const ComponentExample => () => {
 useEffect( () => {
 // Anything in here is fired on component mount.
 }, []);
}
```

If you add a return function inside the useEffect() function, it is triggered when a component unmounts from the DOM.

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

const ComponentExample => () => {
 useEffect(() => {
 return () => {
 // Anything in here is fired on component unmount.
 }
 }, [])
}
```

4.3. PURE COMPONENTS

Q. What are Pure Components in React?

Pure Components in React are the components which do not re-renders when the value of state and props has been updated with the same values. Pure Components restricts the re-rendering ensuring the higher performance of the Component.

Features of React Pure Components:

- Prevents re-rendering of Component if props or state is the same
- Takes care of `shouldComponentUpdate()` implicitly
- `State()` and `Props` are Shallow Compared
- Pure Components are more performant in certain cases

Example:

```
/**
 * React Pure Component
 */
import React from "react";

export default class App extends React.PureComponent {
 constructor() {
 super();
 this.state = {
 userArray: [1, 2, 3, 4, 5]
 };
 // Here we are creating the new Array Object during setState using "Spread" Operator
 setInterval(() => {
 this.setState({
```

```
 userArray: [...this.state.userArray, 6]
 });
 }, 1000);
 }

 render() {
 return <b>Array Length is: {this.state.userArray.length}</b>;
 }
}
```

↥

Q. What is difference between Pure Component vs Component?

PureComponent is exactly the same as Component except that it handles the `shouldComponentUpdate()` method. The major difference between React.PureComponent and React.Component is PureComponent does a shallow comparison on state change. It means that when comparing scalar values it compares their values, but when comparing objects it compares only references. It helps to improve the performance of the app.
A component rerenders every time its parent rerenders, regardless of whether the component's props and state have changed. On the other hand, a pure component will not rerender if its parent rerenders, unless the pure component's `props` (or `state`) have changed.
When to use React.PureComponent:

- State/Props should be an immutable object
- State/Props should not have a hierarchy
- We should call forceUpdate when data changes

Example:

```
// Regular class component
class App extends React.Component {
 render() {
 return <h1>Component Example !</h1>
 }
}

// React Pure class component
class Message extends React.Component {
 render() {
 return <h1>PureComponent Example !</h1>
 }
}
```

↥

Q. What are the problems of using render props with PureComponent?

If you create a function inside a **render** method, it negates the purpose of pure component. Because the shallow prop comparison will always return **false** for new props, and each **render** in this case will generate a new value for the render prop. You can solve this issue by defining the render function as instance method.

Example:

```
class Mouse extends React.PureComponent {
 // Mouse Component...
}

class MouseTracker extends React.Component {
  // Defined as an instance method, `this.renderTheCat` always
 // refers to *same* function when we use it in render
 renderTheCat(mouse) {
```

```
 return <Cat mouse={mouse} />;
 }

 render() {
 return (
 <div>
 <h1>Move the mouse around!</h1>
 {/* define the render function as instance method */}
 <Mouse render={this.renderTheCat} />
 </div>
 );
 }
}
```

↥

Q. When to use PureComponent over Component?

- We want to avoid re-rendering cycles of component when its props and state are not changed
- The state and props of component are immutable
- We do not plan to implement own `shouldComponentUpdate()` lifecycle method.

On the other hand, we should not use `PureComponent()` as a base component if:

- props or state are not immutable
- Plan to implement own `shouldComponentUpdate()` lifecycle method.

↥

4.4. HIGHER ORDER COMPONENTS

Q. What is Higher Order Components in React.js?

A **Higher-Order Component(HOC)** is a function that takes a component and returns a new component. It is the advanced technique in React.js for reusing a component logic.

Higher-Order Components are not part of the React API. They are the pattern that emerges from React's compositional nature. The component transforms props into UI, and a higher-order component converts a component into another component. The examples of HOCs are Redux's connect and Relay's createContainer.

```
/**
 * Higher Order Component
 */
import React, { Component } from "react";

export default function Hoc(HocComponent) {
 return class extends Component {
 render() {
 return (
 <div>
 <HocComponent></HocComponent>
 </div>
 );
 }
 };
}

/**
 * App.js
 */
import React, { Component } from "react";
import Hoc from "./HOC";
export default class App extends Component {
 render() {
 return <h2>Higher Order Component!</h2>;
 }
```

```
}
App = Hoc(App);
```

Note:

- A HOC do not modify or mutate components. It creates a new ones.
- A HOC is used to compose components for code reuse.
- A HOC is a pure function. It has no side effects, returning only a new component.

↥

Q. What are the benefits of using HOC?

Benefits:

- Importantly they provided a way to reuse code when using ES6 classes.
- No longer have method name clashing if two HOC implement the same one.
- It is easy to make small reusable units of code, thereby supporting the single responsibility principle.
- Apply multiple HOCs to one component by composing them. The readability can be improve using a compose function like in Recompose.

Problems:

- Boilerplate code like setting the **displayName** with the HOC function name e.g. (`withHOC(Component)`) to help with debugging.
- Ensure all relevant props are passed through to the component.
- Hoist static methods from the wrapped component.
- It is easy to compose several HOCs together and then this creates a deeply nested tree making it difficult to debug. ↥

Q. What are Higher Order Component factory implementations?

Creating a higher order component basically involves manipulating WrappedComponent which can be done in two ways:

- Props Proxy
- Inheritance Inversion

Both enable different ways of manipulating the WrappedComponent.

1. Props Proxy:

In this approach, the render method of the HOC returns a React Element of the type of the WrappedComponent. We also pass through the props that the HOC receives, hence the name **Props Proxy**.

Example:

```
function ppHOC(WrappedComponent) {
 return class PP extends React.Component {
 render() {
 return <WrappedComponent {...this.props}/>
 }
 }
}
```

Props Proxy can be implemented via a number of ways

- Manipulating props
- Accessing the instance via Refs
- Abstracting State
- Wrapping the WrappedComponent with other elements

2. Inheritance Inversion:

Inheritance Inversion allows the HOC to have access to the WrappedComponent instance via `this` keyword, which means it has access to the `state`, `props`, component lifecycle hooks and the `render` method.
Example:

```
function iiHOC(WrappedComponent) {
 return class Enhancer extends WrappedComponent {
 render() {
 return super.render()
 }
 }
}
```

Inheritance Inversion can be used in:

- Conditional Rendering (Render Highjacking)
- State Manipulation

↥

Q. Explain Inheritance Inversion (iiHOC) in react?

Inheritance Inversion gives the HOC access to the WrappedComponent instance via this, which means we can use the `state`, `props`, component lifecycle and even the `render` method.
Example:

```
/**
 * Inheritance Inversion
 */
class Welcome extends React.Component {
 render() {
 return (
 <div> Welcome {his.props.user}</div>
 )
 }
}

const withUser = (WrappedComponent) => {
 return class extends React.Component {
 render() {
 if(this.props.user) {
 return (
 <WrappedComponent {...this.props} />
 )
 }
 return <div>Welcome Guest!</div>
 }
 }
}
const withLoader = (WrappedComponent) => {
 return class extends WrappedComponent {
 render() {
 const { isLoader } = this.props
 if(!isLoaded) {
 return <div>Loading...</div>
 }
 return super.render()
 }
 }
```

```
}

export default withLoader(withUser(Welcome))
```

↥

Q. How to create props proxy for Higher Order Component component?

It's nothing more than a function, propsProxyHOC, that receives a Component as an argument (in this case we've called the argument WrappedComponent) and returns a new component with the WrappedComponent within.

When we return the Wrapped Component we have the possibility to manipulate props and to abstract state, even passing state as a prop into the Wrapped Component.

We can create `props` passed to the component using props proxy pattern as below

```
const propsProxyHOC = (WrappedComponent) => {

 return class extends React.Component {
 render() {
 const newProps = {
 user: currentLoggedInUser
 }

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

Props Proxy HOCs are useful to the following situations:

- Manipulating props
- Accessing the instance via Refs (be careful, avoid using refs)
- Abstracting State
- Wrapping/Composing the WrappedComponent with other elements

↥

Q. How to use decorators in React?

Decorators provide a way of calling Higher-Order functions. It simply take a function, modify it and return a new function with added functionality. The key here is that they don't modify the original function, they simply add some extra functionality which means they can be reused at multiple places.

Example:

```
export const withUniqueId = (Target) => {
 return class WithUniqueId extends React.Component {
 uid = uuid();

 render() {
 return <Target {...this.props} uuid={this.uid} />;
 }
 };
}

@withUniqueId
class UniqueIdComponent extends React.Component {
 render() {
 return <div>Generated Unique ID is: {this.props.uuid}</div>;
 }
}

const App = () => (
 <div>
 <h2>Decorators in React!</h2>
 <UniqueIdComponent />
```

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

Note: Decorators are an experimental feature in React that may change in future releases.

☆

↥

Q. What is the purpose of displayName class property?

The **displayName** string is used in debugging messages. Usually, you don't need to set it explicitly because it's inferred from the name of the function or class that defines the component. You might want to set it explicitly if you want to display a different name for debugging purposes or when you create a higher-order component.

Example:

```
function withSubscription(WrappedComponent) {

 class WithSubscription extends React.Component {/* ... */}

 WithSubscription.displayName = `WithSubscription(${getDisplayName(WrappedComponent)})`;
 return WithSubscription;
}

function getDisplayName(WrappedComponent) {
 return WrappedComponent.displayName || WrappedComponent.name || 'Component';
}
```

↥

4.5. LAZY LOADING

Q. How to set up lazy loading components in React? 1. REACT.LAZY():

React.lazy is a function that lets you load components lazily through what is called code splitting without help from any external libraries. It makes possible for us to dynamically import components but they are rendered like regular components. This means that the bundle containing the component will only be loaded when the component is rendered.

React.lazy() takes a function that returns a promise as it's argument, the function returns a promise by calling import() to load the content. The returned Promise resolves to a module with a default containing the React Component.

```
// Without Lazy
import MyComponent from './MyComponent';

// With Lazy
const MyComponent = React.lazy(() => import('./MyComponent'));
```

2. SUSPENSE:
React.Suspense is a component that can be used to wrap lazy components. A React.Suspense takes a fallback prop that can be any react element, it renders this prop as a placeholder to deliver a smooth experience and also give user feedback while the lazy component is being loaded.

```
/**
 * Suspense
 */
import React, { Suspense } from 'react';

const MyComponent = React.lazy(() => import('./MyComponent'));
```

```
const App = () => {
 return (
 <div>
 <Suspense fallback={<div>Loading ... </div>}>
 <MyComponent />
 </Suspense>
 </div>
 );
}
```

Example:

```
/**
 * React Lazy Loading Routes
 */
import React, { Suspense, lazy } from "react";
import { Switch, BrowserRouter as Router, Route, Link } from "react-router-dom";

const Home = lazy(() => import("./Home"));
const ContactUs = lazy(() => import("./ContactUs"));
const HelpPage = lazy(() => import("./Help"));

export default function App() {
 return (
 <Router>
 <ul>
 <li><Link to="/">Home</Link></li>
 <li><Link to="/contact-us">ContactUs</Link></li>
 <li><Link to="/help">HelpPage</Link></li>
 </ul>
 <hr />
 <Suspense fallback={<h1>Loading...</h1>}>
 <Switch>
 <Route exact component={Home} path="/" />
 <Route component={ContactUs} path="/contact-us" />
 <Route component={HelpPage} path="/help" />
 </Switch>
 </Suspense>
 </Router>
 );
}
```

☆

↥

5. REACT PROPS

Q. What is props in React?

Props is a special keyword in React, which stands for properties and is being used for passing data from one component to another. However, callback functions can also be passed, which can be executed inside the child to initiate an update.

Props are **immutable** so we cannot modify the props from inside the component. These attributes are available in the class component as **this.props** and can be used to render dynamic data in our render method.
Example:

```
function Welcome(props) {
 return <h1>Hello, {props.name}</h1>;
}

const element = <Welcome name="World!" />;
```

☆

↥

Q. Why props in React are read only?

When you declare a component as a function or a class, it must never modify its own props.

Consider this sum function:

```
function sum(a, b) {
 return a + b;
}
```

Such functions are called **pure** because they do not attempt to change their inputs, and always return the same result for the same inputs. All React components must act like pure functions with respect to their props. A component should only manage its own state, but it should not manage its own props.

In fact, props of a component is concretely "the state of the another component (parent component)". So props must be managed by their component owner. That's why all React components must act like pure functions with respect to their props (not to mutate directly their props).

☆

↥

Q. What are default props?

The defaultProps is a React component property that allows you to set default values for the props argument. If the prop property is passed, it will be changed.

The `defaultProps` can be defined as a property on the component class itself to set the default props for the class. `defaultProps` is used for **undefined** props, not for **null** props.

```
/**
 * Default Props
 */
class MessageComponent extends React.Component {
 render() {
 return (
 <div>Hello, {this.props.value}.</div>
 )
 }
}

// Default Props
MessageComponent.defaultProps = {
 value: 'World'
}

ReactDOM.render(
 <MessageComponent />,
 document.getElementById('default')
)

ReactDOM.render(
 <MessageComponent value='Folks'/>,
 document.getElementById('custom')
)
```

☆

↥

Q. How to access props inside quotes in React JSX?

React JSX doesn't support variable interpolation inside an attribute value, but we can put any JS expression inside curly braces as the entire attribute value.

Approach 1: Putting js expression inside curly braces

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

Approach 2: Using ES6 template literals.

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

Example:

```
/**
 * Access Props
 */
class App extends Component {
 render() {
 return (
 <div>
 <img
 alt="React Logo"

 // Using ES6 template literals
 src={`${this.props.image}`}
 />
 </div>
 );
 }
}
 export default App;
```

↥

Q. How to pass numbers to React component?

In react, numbers can be passed via curly braces(`{}`) where as strings in quotes (`""`);
Example:

```
function App() {
 return <Greetings name="Nathan" age={27} occupation="Software Developer" />;
}

// Greetings Component
function Greetings(props) {
 return (
 <h2>
 Hello! I'm {props.name}, a {props.age} years old {props.occupation}.
 Pleased to meet you!
 </h2>
 );
}
```

↥

Q. How are boolean props used in React?

React JSX has exactly two ways of passing true, `<MyComponent prop />` and `<MyComponent prop={true} />` and exactly one way of passing false `<MyComponent prop={false} />`.
Example:

```
/**
 * Boolean Props
 */
const MyComponent = ({ prop1, prop2 }) => (
 <div>
 <div>Prop1: {String(prop1)}</div>
 <div>Prop2: {String(prop2)}</div>
 </div>
)

function App() {
 return (
 <div>
 <MyComponent prop1={true} prop2={false} />
 <MyComponent prop1 prop2 />
 <MyComponent prop1={false} prop2 />
```

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

↥

Q. How to apply validation on Props in React?

Props are an important mechanism for passing the **read-only** attributes to React components. React provides a way to validate the props using `PropTypes`. This is extremely useful to ensure that the components are used correctly.
Example:

```
/**
 * Props Validation
 */
import React from "react";
import PropTypes from "prop-types";

export default class App extends React.Component {
 render() {
 return (
 <>
 <h3>Boolean: {this.props.propBool ? "True" : "False"}</h3>
 <h3>Array: {this.props.propArray}</h3>
 <h3>Number: {this.props.propNumber}</h3>
 <h3>String: {this.props.propString}</h3>
 </>
 );
 }
}

App.defaultProps = {
 propBool: true,
 propArray: [10, 20, 30],
 propNumber: 100,
 propString: "Hello React!"
};

App.propTypes = {
 propBool: PropTypes.bool.isRequired,
 propArray: PropTypes.array.isRequired,
 propNumber: PropTypes.number,
 propString: PropTypes.string
};
```

↥

Q. How to specify the shape of an object with PropTypes

The `PropTypes.shape()` validator can be used when describing an object whose keys are known ahead of time, and may represent different types.
Example:

```
/**
 * PropTypes.shape()
 * @param {*} props
 */
import PropTypes from "prop-types";

const Component = (props) => (
 <div>
 Component badge: {props.badge ? JSON.stringify(props.badge) : "none"}
 </div>
);

// PropTypes validation for the prop object
Component.propTypes = {
 badge: PropTypes.shape({
 src: PropTypes.string.isRequired,
```

```
 alt: PropTypes.string.isRequired
 })
};

const App = () => (
 <div>
 <Component badge={{ src: "horse.png", alt: "Running Horse" }} />
 {/*<Component badge={{src:null, alt: 'this one gives an error'}}/>*/}
 <Component />
 </div>
);

export default App;
```

Output:

```
Component badge: {"src":"horse.png","alt":"Running Horse"}
Component badge: none
```

☆

↥

Q. How PropTypes.objectOf is different from PropTypes.shape?

The `PropTypes.objectOf()` validator is used when describing an object whose keys might not be known ahead of time, and often represent the same type.
Example:

```
/**
 * PropTypes
 */
import PropTypes from 'prop-types';

// Expected prop object - dynamic keys (i.e. user ids)
const myProp = {
 25891102: 'Shila Jayashri',
 34712915: 'Employee',
 76912999: 'shila.jayashri@email.com'
};

// PropTypes validation for the prop object
MyComponent.propTypes = {
 myProp: PropTypes.objectOf(PropTypes.number)
};
```

↥

Q. How React PropTypes allow different types for one prop?

Using `PropTypes.oneOfType()` says that a prop can be one of any number of types. For instance, a phone number may either be passed to a component as a string or an integer:

```
/**
 * PropTypes.oneOfType()
 */
const Component = (props) => <div>Phone Number: {props.phoneNumber}</div>

Component.propTypes = {
 phoneNumber: PropTypes.oneOfType([
 PropTypes.number,
 PropTypes.string
 ]),
}

const App = () => (
 <div>
 <Component phoneNumber={04403472916}/>
 {/*<Component phoneNumber={"2823788557"}/>*/}
```

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

☆

↥

Q. What are render props?

The term **render props** refers to a technique for sharing code between React components using a prop whose value is a function.

In simple words, render props are simply props of a component where you can pass functions. These functions need to return elements, which will be used in rendering the components.

Example:

```
/**
 * Render Props
 */
import React from "react";
import Wrapper from "./Wrapper";

class App extends React.Component {
 render() {
 return (
 <Wrapper
 render={({ increment, count }) => (
 <div>
 <h3>Render Props Counter</h3>
 <p>{count}</p>
 <button onClick={() => increment()}>Increment</button>
 </div>
 )}
 />
 );
 }
}

/**
 * Wrapper Component
 */
class Wrapper extends React.Component {
 state = {
 count: 0
 };
 // Increase count
 increment = () => {
 const { count } = this.state;
 return this.setState({ count: count + 1 });
 };

 render() {
 const { count } = this.state;

 return (
 <div>
 {this.props.render({ increment: this.increment, count: count })}
 </div>
 );
 }
}
```

☆

↥

Q. What are the benefits of using Render Props? **Benefits:**

- Reuse code across components when using ES6 classes.

- The lowest level of indirection - it's clear which component is called and the state is isolated.
- No naming collision issues for props, state and class methods.
- No need to deal with boiler code and hoisting static methods.

Problems:

- Caution using `shouldComponentUpdate()` as the render prop might close over data it is unaware of. • There could also be minor memory issues when defining a closure for every render. But be sure to measure first before making performance changes as it might not be an issue for your app.
- Another small annoyance is the render props callback is not so neat in JSX as it needs to be wrapped in an expression. Rendering the result of an HOC does look cleaner.

↥

Q. How do you create Higher Order Component using render props?

It is possible to implement most higher-order components (HOC) using a regular component with a render prop. This way render props gives the flexibility of using either pattern.

Example:

```
function withMouse(Component) {
 return class extends React.Component {
 render() {
  return (
 <Mouse render={mouse => (
 <Component {...this.props} mouse={mouse} />
 )}/>
 );
 }
 }
}
```

↥

Q. Explain HOC vs render props in react.js?

The Higher-Order Components, Render Props and Hooks are three patterns to implement **state-** or *behaviour-** sharing between components. All three have their own use cases and none of them is a full replacement of the others.

1. Higher-order components:

Essentially HOC are similar to the decorator pattern, a function that takes a component as the first parameter and returns a new component. This is where you apply your crosscutting functionality.

Example:

```
function withExample(Component) {
 return function(props) {
 // cross cutting logic added here
 return <Component {...props} />;
 };
}
```

2. Render Props:

A render prop is where a component's prop is assigned a function and this is called in the render method of the component. Calling the function can return a React element or component to render.

Example:

```
render(){
 <FetchData render={(data) => {
```

```
 return <p>{data}</p>
 }} />
}
```

The React community is moving away from HOC (higher order components) in favor of render prop components (RPC). For the most part, HOC and render prop components solve the same problem. However, render prop components provide are gaining popularity because they are more declarative and flexible than an HOC.

↥

Q. What is children props?

The `{this.props.children}` is a special prop, automatically passed to every component, that can be used to render the content included between the opening and closing tags when invoking a component.
Example:

```
/**
 * React Children Props
 */
class MyComponent extends React.Component {
 render() {
 return (
 <div>
 <h1>React Children Props Example</h1>
 {this.props.children}
 </div>
 );
 }
}

class OtherComponent extends React.Component {
 render() {
 return <div>Other Component Props</div>;
 }
}

ReactDOM.render(
 <MyComponent>
 <p>React DOM Props</p> {/* Children Props*/}
 <OtherComponent />
 </MyComponent>,
 document.getElementById("root")
);
```

↥

Q. When we should use React.cloneElement vs this.props.children?

The `React.cloneElement()` works if child is a single React element.
For almost everything `{this.props.children}` is used. Cloning is useful in some more advanced scenarios, where a parent sends in an element and the child component needs to change some props on that element or add things like `ref` for accessing the actual DOM element.
React.Children:

Since `{this.props.children}` can have one element, multiple elements, or none at all, its value is respectively a single child node, an array of child nodes or undefined. Sometimes, we want to transform our children before rendering them — for example, to add additional props to every child. If we wanted to do that, we'd have to take the possible types of `this.props.children` into account. For example, if there is only one child, we can not map it.
Example:

```
/**
 * React Children Props
 */
export default class App extends React.Component {
 render() {
 return (
```

```
 <div>
 <b>Children ({this.props.children.length}):</b>
 {this.props.children}
 </div>
 );
 }
}

class Widget extends React.Component {

 render() {
 return
 <div>
 <h2>First Example:</h2>
 <App>
 <div>10</div>
 <div>20</div>
 <div>30</div>
 </App>
 <h2>Second Example:</h2>
 <App>
 <div>A</div>
 <div>B</div>
 </App>
 </div>
 }
}
```

Output

```
First Example:
Children (3):
10
20
30

Second Example:
Children (2):
A
B
```

☆

↥

Q. What do these three dots in React do?

The ES6 Spread operator or Rest Parameters is use to pass `props` to a React component. Let us take an example for a component that expects two props:

```
function App() {
 return <Hello firstName="Pallav" lastName="Hegde" />
}
```

Using the Spread operator, it become like this

```
function App() {
 const props = {firstName: 'Pallav', lastName: 'Hegde'}
 return <Hello {...props} />
}
```

When we use the `...props` syntax, actaully it expand the props object from the parent component, which means all its attributes are passed down the child component that may not need them all. This will make things like debugging harder. **Using the Spread Operator with setState() for Setting the Nested State:**

Let us suppose we have a state with a nested object in our component:

```
this.state = {
 stateObj: {
 attr1: '',
 attr2: '',
 },
}
```

We can use the Spread syntax to update the nested state object.

```
this.setState(state => ({
 person: {
```

```
 ...state.stateObj,
 attr1: 'value1',
 attr2: 'value2',
 },
}))
```

↥

Q. 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.

Problem: This will to add the unknown HTML attribute `flag` to the DOM element.

```
const Sample = () => (<Spread flag={true} className="content"/>);
const Spread = (props) => (<div {...props}>Test</div>);
```

Solution: By creating props specifically for DOM attribute, we can safely spread.

```
const Sample = () => (<Spread flag={true} domProps={{className: "content"}}/>);
const Spread = (props) => (<div {...props.domProps}>Test</div>);
```

Or alternatively we can use prop destructuring with `...rest`:

```
const Sample = () => (<Spread flag={true} className="content"/>);
const Spread = ({ flag, ...domProps }) => (<div {...domProps}>Test</div>);
```

Note:

In scenarios where you use a PureComponent, when an update happens it re-renders the component even if domProps did not change. This is because PureComponent only shallowly compares the objects.

↥

Q. What will happen if you use props in initial state?

Using props to generate state in `getInitialState` often leads to duplication of "source of truth", i.e. where the real data is. This is because getInitialState is only invoked when the component is first created.
The danger is that 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 (or getInitialState) will never update the current state of the component. The initialization of state from `props` only runs when the component is first created.
Bad:

The below component won't display the updated input value

```
class App extends React.Component {

 // constructor function (or getInitialState)
 constructor(props) {
 super(props)

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

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

Good:

Using props inside render method will update the value:

```
class App extends React.Component {

 // constructor function (or getInitialState)
 constructor(props) {
 super(props)

 this.state = {
```

```
 records: []
 }
 }

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

↥

Q. 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. The `React.cloneElement()` function returns a copy of a specified element. Additional props and children can be passed on in the function. We shoul use this function when a parent component wants to add or modify the `props` of its children. `import React from 'react'`

```
export default class App extends React.Component {
 // rendering the parent and child component
 render() {
 return (
 <ParentComp>
 <MyButton/>
 <br/>
 <MyButton/>
 </ParentComp>
 )
 }
}

/**
 * The parent component
 */
class ParentComp extends React.Component {
 render() {
 // The new prop to the added.
 let newProp = 'red'
 // Looping over the parent's entire children,
 // cloning each child, adding a new prop.
 return (
 <div>
 {React.Children.map(this.props.children,
 child => {
 return React.cloneElement(child,
 {newProp}, null)
 })}
 </div>
 )
 }
}

/**
 * The child component
 */
class MyButton extends React.Component {
 render() {
 return <button style =
 {{ color: this.props.newProp }}>
 Hello World!</button>
 }
}
```

↥

Q. When should I be using React.cloneElement vs this.props.children?

The `React.cloneElement` only works if your child is a single React element.
Example:

```
<ReactCSSTransitionGroup
 component="div"
 transitionName="example"
 transitionEnterTimeout={500}
 transitionLeaveTimeout={500}
 >
 {React.cloneElement(this.props.children, {
 key: this.props.location.pathname
 })}
</ReactCSSTransitionGroup>
```

For almost everything `{this.props.children}` is used. Cloning is useful in some more advanced scenarios, where a parent sends in an element and the child component needs to change some props on that element or add things like `ref` for accessing the actual DOM element.
Example:

```
class Users extends React.Component {
 render() {
 return (
 <div>
 <h2>Users</h2>
 {this.props.children}
 </div>
 )
 }
}
```

Q. How to pass JSON Objects from Child to Parent Component?

Example: Passing JSON Objects from Child to Parent Component using callback function

```
// Parent Component

export default class App extends React.Component {
 constructor() {
 super();
 this.state = {
 message: ""
 };
 this.onSubmitMessage = this.onSubmitMessage.bind(this);
 }

 onSubmitMessage(message) {
 this.setState({ message: message });
 }

 render() {
 const { message } = this.state;

 return (
 <div>
 <h3>Parent component</h3>
 <div>The message coming from the child component is : {message}</div>
 <hr />
 <Child
 // passing as callback function
 onSubmitMessage={this.onSubmitMessage}
 />
 </div>
 );
 }
}

// Child Component

export default class Child extends React.Component {
 constructor() {
 super();
 this.state = {
```

```
 greetingMessag: ""
 };
 this.onMessageChange = this.onMessageChange.bind(this);
 this.onSubmit = this.onSubmit.bind(this);
 }

 onMessageChange(event) {
 let message = event.target.value;
 this.setState({ greetingMessag: message });
 }

 // pass message to parent component using callback
 onSubmit() {
 this.props.onSubmitMessage(this.state.greetingMessag);
 }

 render() {
 return (
 <div>
 <h3>Child Component</h3>
 <input
 type="text"
 onChange={this.onMessageChange}
 placeholder="Enter a message"
 />
 <button onClick={this.onSubmit}>Submit</button>
 </div>
 );
 }
}
```

↥

Q. What is the use of this props?

It is called spread operator (ES6 feature) and its aim is to make the passing of props easier.

Example:

```
<div {...this.props}>
 Content Here
</div>
```

It is equal to Class Component

```
const person = {
 name: "Alex",
 age: 26,
 coun : "India"
}

class SpreadExample extends React.Component {
 render() {
 const {name, age, coun } = {...this.props}
 return (
 <div>
 <h3> Person Information: </h3>
 <ul>
 <li>name={name}</li>
 <li>age={age}</li>
 <li>coun ={coun }</li>
 </ul>
 </div>
 )
 }
}

ReactDOM.render(
 <SpreadExample {...person}/>
 , mountNode
)
```

↥

6. REACT STATE

Q. What is State in React?

The state is a built-in object that is used to contain data about the component. A component's state can change over time; whenever it changes, the component re-renders.

Example:

```
/**
 * React State
 */
export default class Employee extends React.Component {
 constructor() {
 super();
 this.state = {
 id: 100,
 name: "Sarita Mangat"
 };
 }

 render() {
 return (
 <div>
 <div>ID: {this.state.id}</div>
 <div>Name: {this.state.name}</div>
 </div>
 );
 }
}
```

↥

Q. What does setState() do?

The component state can be updated in response to event handlers, server responses, or prop changes. This is done using the **setState()** method. The setState() method enqueues all of the updates made to the component state and instructs React to re render the component and its children with the updated state.

Always use the setState() method to change the state object, since it will ensure that the component knows it's been updated and calls the `render()` method.

Example:

```
/**
 * React setState()
 */
export default class App extends React.Component {
 constructor(props) {
 super(props);

 this.state = {
 color: "blue"
 };
 }

 handleColor = () => {
 this.setState({ color: "red" });
 };

 render() {
 return (
 <div>
 <h3 style={{ color: `${this.state.color}` }}>
 Color: {this.state.color}
```

```
</h3>
<button type="button" onClick={this.handleColor}>
Change color
</button>
</div>
);
}
}
```

☆

↥

Q. Why is setState() in React async instead of sync?

The **setState()** does not immediately mutate `this.state()` but creates a pending state transition. Accessing `this.state()` after calling this method can potentially return the existing value. There is no guarantee of synchronous operation of calls to setState() and calls may be batched for performance gains.
This is because setState() alters the state and causes rerendering. This can be an expensive operation and making it synchronous might leave the browser unresponsive. Thus the setState() calls are asynchronous as well as batched for better UI experience and performance.

↥

Q. What is the second argument that can optionally be passed to setState() and what is its purpose?

A **callback function** which will be invoked when `setState()` has finished and the component is re-rendered. The setState() is asynchronous, which is why it takes in a second callback function. Typically it's best to use another lifecycle method rather than relying on this callback function, but it is good to know it exists.
Example:

```
this.setState(
 { username: 'Lila' },
 () => console.log('setState has finished and the component has re-rendered.')
)
```

The setState() will always lead to a re-render unless `shouldComponentUpdate()` returns **false**. To avoid unnecessary renders, calling setState() only when the new state differs from the previous state makes sense and can avoid calling setState() in an infinite loop within certain lifecycle methods like `componentDidUpdate()`.

↥

Q. What are the possible ways of updating objects in state?

Instead of directly modifying the state using `this.state()`, we use `this.setState()`. This is a function available to all React components that use state, and allows us to let React know that the component state has changed. This way the component knows it should re-render, because its state has changed and its UI will most likely also change.
Example:

```
this.state = {
 user: { name: 'Vasuda Handa', age: 22 }
}
```

- **Using Object.assign()**

```
this.setState(prevState => {
 let user = Object.assign({}, prevState.user); // creating copy of state variable user
 user.name = 'Sai Gupta'; // update the name property, assign a new value  return { user }; // return
new object user object
})
```

- **Using spread syntax**

```
this.setState(prevState => ({
 user: { // object that we want to update
 ...prevState.user, // keep all other key-value pairs
 name: 'Niraj Gara' // update the value of specific key
 }
}))
```

☆

↥

Q. What will happen if you use setState() in constructor?

When we use `setState()`, then apart from assigning to the object state react also rerenders the component and all it's children. Which we don't need in the constructor, since the component hasn't been rendered anyway.
Inside constructor uses `this.state = {}` directly, other places use `this.setState({ })`
Example:

```
import React, { Component } from 'react'

class Food extends Component {

 constructor(props) {
 super(props)

 this.state = {
 fruits: ['apple', 'orange'],
 count: 0
 }
 }
 render() {
 return (
 <div className = "container">
 <h2> Hello!!!</h2>
 <p> I have {this.state.count} fruit(s)</p>
 </div>
 )
 }
}
```

☆

↥

Q. Why should not we update the state directly?

The **setState()** does not immediately mutate `this.state()` but creates a pending state transition. Accessing `this.state()` after calling this method can potentially return the existing value.
The setState() will always trigger a re-render unless conditional rendering logic is implemented in **shouldComponentUpdate()**. If mutable objects are being used and the logic cannot be implemented in shouldComponentUpdate(), calling setState() only when the new state differs from the previous state will avoid unnecessary re-renders.

Example:

```
import React, { Component } from 'react'

class App extends Component {
 constructor(props) {
 super(props)

 this.state = {
 list: [
 { id: '1', age: 42 },
 { id: '2', age: 33 },
 { id: '3', age: 68 },
 ],
 }
```

```
}

onRemoveItem = id => {
this.setState(state => {
const list = state.list.filter(item => item.id !== id)

return {
list,
}
})
}

render() {
return (
<div>
<ul>
{this.state.list.map(item => (
<li key={item.id}>
The person is {item.age} years old.
<button
type="button"
onClick={() => this.onRemoveItem(item.id)}
>
Remove
</button>
</li>
))}
</ul>
</div>
)
}
}

export default App
```

↥

Q. How to delete an item from state array?

When using React, we should never mutate the state directly. If an object is changed, we should create a new copy. The better approach is to use `Array.prototype.filter()` method which creates a new array.
Example:

```
onDeleteByIndex(index) {
 this.setState({
 users: this.state.users.filter((item, i) => i !== index)
 });
}
```

↥

Q. Why should not call setState() in componentWillUnmount()?

We should not call `setState()` in `componentWillUnmount()` because the component will never be re-rendered. Once a component instance is unmounted, it will never be mounted again.
The `componentWillUnmount()` is invoked immediately before a component is unmounted and destroyed. This method can be used to perform any necessary cleanup method, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in `componentDidMount()`.

↥

Q. How can you re-render a component without using

setState() function?

React components automatically re-render whenever there is a change in their state or props. A simple update of the state, from anywhere in the code, causes all the User Interface (UI) elements to be re-rendered automatically.

However, there may be cases where the render() method depends on some other data. After the initial mounting of components, a re-render will occur.

Using forceUpdate():

The following example generates a random number whenever it loads. Upon clicking the button, the `forceUpdate()` function is called which causes a new, random number to be rendered:

```
/**
 * forceUpdate()
 */
export default class App extends React.Component {
 constructor(){
 super();
 this.forceUpdateHandler = this.forceUpdateHandler.bind(this);
 };

 forceUpdateHandler(){
 this.forceUpdate();
 };

 render(){
 return(
 <div>
 <button onClick= {this.forceUpdateHandler} >FORCE UPDATE</button>
 <h4>Random Number : { Math.random() }</h4>
 </div>
 );
 }
}
```

☆

Note: We should to avoid all uses of `forceUpdate()` and only read from `this.props` and `this.state` in render(). ↥

Q. 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 we should not rely on the current state when calling `setState()`.
The solution is to **pass a function to setState()**, with the previous state as an argument. By doing this we can avoid issues with the user getting the old state value on access due to the asynchronous nature of `setState()`.
Problem:

```
// 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
```

☆

Solution:

```
this.setState((prevState) => ({
 count: prevState.count + 1
}));
this.setState((prevState) => ({
 count: prevState.count + 1
}));
this.setState((prevState) => ({
 count: prevState.count + 1
}));
// this.state.count === 3 as expected
```

☆

↥

Q. How to update nested state properties in React.js?

We can pass the old nested object using the spread operator and then override the particular properties of the nested object. **Example:**

```
// Nested object
state = {
 name: 'Vyasa Agarwal',
 address: {
 colony: 'Old Cross Rds, Mehdipatnam',
 city: 'Patna',
 state: 'Jharkhand'
 }
};

 handleUpdate = () => {
 // Overriding the city property of address object
 this.setState({ address: { ...this.state.address, city: "Ranchi" } })
}
```

☆

↥

Q. What is the difference between state and props?

Props State

Props are read-only. State changes can be asynchronous. Props are immutable. State is mutable.

Props allow you to pass data from one component to other components as an argument. State holds information about the components. Props can be accessed by the child component. State cannot be accessed by child components.

Props are used to communicate between components. States can be used for rendering dynamic changes with the component.

Stateless component can have Props. Stateless components cannot have State. Props make components reusable. State cannot make components reusable.

Props are external and controlled by whatever renders the component. The State is internal and controlled by the React Component itself.

↥

Q. 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.

```
inputChangeHandler : function (event) {
 this.setState({ [event.target.id]: event.target.value });

 // alternatively using template strings for strings
```

```
 // this.setState({ [`key${event.target.id}`]: event.target.value });
}
```

☆

↥

Q. How to listen state change in React.js?

The following lifecycle methods will be called when state changes. You can use the provided arguments and the current state to determine if something meaningful changed.

```
componentWillUpdate(object nextProps, object nextState)
componentDidUpdate(object prevProps, object prevState)
```

In functional component, listen state changes with useEffect hook like this

```
export function MyComponent(props) {
 const [myState, setMystate] = useState('initialState')

 useEffect(() => {
 console.log(myState, '- Has changed')
 },[myState]) // <-- here put the parameter to listen
}
```

↥

Q. How to access child's state in React?

Using Refs:

In React we can access the child's state using `React.createRef()`. We will assign a Refs for the child component in the parent component, then using Refs we can access the child's state.

```
// App.js

class App extends React.Component {
 constructor(props) {
 super(props);
 this.ChildElement = React.createRef();
 }
 handleClick = () => {
 const childelement = this.ChildElement.current;
 childelement.getMsg("Message from Parent Component!");
 };
 render() {
 return (
 <div>
 <Child ref={this.ChildElement} />
 <button onClick={this.handleClick}>CLICK ME</button>
 </div>
 );
 }
}

// Child.js

class Child extends React.Component {
 state = {
 name: "Message from Child Component!"
 };
 getMsg = (msg) => {
 this.setState({
 name: msg
 });
 };
 render() {
 return <h2>{this.state.name}</h2>;
 }
}
```

☆

↥

Q. How to change the state of a child component from its parent in React?

To change child component's state from parent component with React, we can pass props.

```
/**
 * Change Child state from its Parent
 * @param {*} param0
 */
const Child = ({ open }) => {
 return <h2>Child State: {open.toString()}</h2>;
};

const Parent = () => {
 const [isOpen, setIsOpen] = React.useState(false);

 const toggleChild = () => {
 setIsOpen((prevValue) => !prevValue);
 };

 return (
 <div>
 <button onClick={toggleChild}>Click Me</button>
 {/* Pass a callback to Child */}
 <Child open={isOpen} />
 </div>
 );
};

export default Parent;
```

↥

Q. Why is it advised to pass a callback function to setState() as opposed to an object?

Because `this.props` and `this.state` may be updated asynchronously, we should not rely on their values for calculating the next state.
Example: setState Callback in a Class Component

```
import React, { Component } from 'react'

class App extends Component {
 constructor(props) {
 super(props)
 this.state = {
 age: 0,
 }
 }

 // this.checkAge is passed as the callback to setState
 updateAge = (value) => {
 this.setState({ age: value}, this.checkAge)
 }

 checkAge = () => {
 const { age } = this.state
 if (age !== 0 && age >= 21) {
 // Make API call to /beer
 } else {
 // Throw error 404, beer not found
 }
 }

 render() {
 const { age } = this.state
 return (
 <div>
```

```
 <p>Drinking Age Checker</p>
 <input
 type="number"
 value={age}
 onChange={e => this.updateAge(e.target.value)}
 />
 </div>
 )
 }
}
export default App
```

Example: setState Callback in a Functional Component

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

function App() {
 const [age, setAge] = useState(0)

 updateAge(value) {
 setAge(value)
 }

 useEffect(() => {
 if (age !== 0 && age >= 21) {
 // Make API call to /beer
 } else {
 // Throw error 404, beer not found
 }
 }, [age])

 return (
 <div>
 <p>Drinking Age Checker</p>
 <input
 type="number"
 value={age}
 onChange={e => setAge(e.target.value)}
 />
 </div>
 )
}

export default App
```

↥

Q. How does the state differ from props in React? 1. State:

This is data maintained inside a component. It is local or owned by that specific component. The component itself will update the state using the `setState()` function.
Example:

```
class AppComponent extends React.component {
 state = {
 msg : 'Hello World!'
 }

 render() {
 return <div>Message {this.state.msg}</div>
 }
}
```

2. Props:

Data passed in from a parent component. `props` are read-only in the child component that receives them. However, callback functions can also be passed, which can be executed inside the child to initiate an update.
Example: The parent can pass a props by using this

```
<ChildComponent color='red' />
```

Inside the ChildComponent constructor we could access the props

```
class ChildComponent extends React.Component {
```

```
 constructor(props) {
 super(props)
 console.log(props.color)
 }
}
```

Props can be used to set the internal state based on a prop value in the constructor, like this

```
class ChildComponent extends React.Component {
 constructor(props) {
 super(props)
 this.state.colorName = props.color
 }
}
```

Props should never be changed in a child component. Props are also used to allow child components to access methods defined in the parent component. This is a good way to centralize managing the state in the parent component, and avoid children to have the need to have their own state.

Difference between State and Props:

	Props	**State**
	Props are read-only.	State changes can be asynchronous.
	Props allow to pass data from one component to other components as an argument.	State holds information about the components. Props can be accessed by the child component. State cannot be accessed by child components.
	Props are used to communicate between components.	States can be used for rendering dynamic changes with the component.
	Stateless component can have Props.	Stateless components cannot have State.
	Props are external and controlled by whatever renders the component.	The State is internal and controlled by the React Component itself.

↥

7. REACT EVENTS

Q. What is meant by event handling in React? Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:

- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

Example:

```
/**
 * Event Handling in React
 */
export default class Toggle extends React.Component {
 constructor(props) {
```

```
 super(props);
 this.state = { isToggleOn: true };
 }

 handleClick() {
 this.setState((state) => ({
 isToggleOn: !state.isToggleOn
 }));
 }
 render() {
 return (
 <button onClick={() => this.handleClick()}>
 {this.state.isToggleOn ? "ON" : "OFF"}
 </button>
 );
 }
}
```

☆

↥

Q. How to pass a parameter to event handlers in React?

Example:

```
const message = "Hey there!";
export default class App extends React.Component {
 displayMessage(message) {
 alert(message);
 }

 render() {
 return (
 <button onClick={() => this.displayMessage(message)}>CLICK ME</button>
 );
 }
}
```

☆

↥

Q. How do you pass an event handler to a component?

Example:

```
import React, {useState} from "react";
import "./styles.css";

export default function App() {
 return (
 <Container/>
 );
}

const Container = () => {
 const [counter, setCounter] = useState(0);

 const handleCustomClick = () => {
 setCounter(counter + 1)
 }

 return (
  <div>
 <div>Counter: {counter}</div>
 <CustomButton onCustomClick={handleCustomClick}/>
 </div>
 );
```

```
}

const CustomButton = ({onCustomClick}) => {
 return (
 <button onClick={onCustomClick}>
 My Custom Button
 </button>
 );
}
```

☆

↥

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

In HTML, the attribute name is in all lowercase and is given a string invoking a function defined somewhere: `<button onclick="handleClick()"></button>`

In React, the attribute name is camelCase and are passed the function reference inside curly braces:

```
<button onClick={handleClick} />
```

In HTML, `false` can be returned to prevent default behavior, whereas in React `preventDefault()` has to be called explicitly. `<a href="#" onclick="console.log('The link was clicked.'); return false" />`

```
function handleClick(e) {
 e.preventDefault()
 console.log("The link was clicked.")
}
```

↥

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

There are 3 possible ways to achieve this

1. Event Handler in Render Method:

We can bind the handler when it is called in the render method using `bind()` method.

```
handleClick() {
 // ...
}

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

☆

2. Event Handler using Arrow Function:

In this approach we are binding the event handler implicitly. This approach is the best if you want to pass parameters to your event.

```
handleClick() {
 // ...
}

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

☆

3. Event Handler in Constructor:

This has performance benefits as the events aren't binding every time the method is called, as opposed to the previous two approaches.

```
constructor(props) {
```

```
 // This binding is necessary to make `this` work in the callback
 this.handleClick = this.handleClick.bind(this);
}

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

☆

↥

Q. Why do we need to bind methods inside class component constructor?

In Class Components, when we pass the event handler function reference as a callback like this

```
<button type="button" onClick={this.handleClick}>Click Me</button>
```

the event handler method loses its **implicitly bound** context. When the event occurs and the handler is invoked, the `this` value falls back to **default binding** and is set to `undefined`, as class declarations and prototype methods run in strict mode. When we bind the `this` of the event handler to the component instance in the constructor, we can pass it as a callback without worrying about it losing its context.
Arrow functions are exempt from this behavior because they use **lexical** `this` binding which automatically binds them to the scope they are defined in.
Example:

```
/**
 * Event Handling in React
 */
import React from "react";

export default class App extends React.Component {
 constructor(props) {
 super(props);
 this.handleClick = this.handleClick.bind(this);
 }

 handleClick(event) {
 alert("Click event triggered!");
 }

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

☆

↥

Q. How do I 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 equivalent to calling `.bind`
```
<button onClick={this.handleClick.bind(this, id)} />
```

Example:

```
/**
 * Pass parameter to an event handler
 */
const A = 65; // ASCII character code
```

```
export default class App extends React.Component {
 constructor(props) {
 super(props);
 this.state = {
 justClicked: null,
 letters: Array.from({ length: 26 }, (_, i) => String.fromCharCode(A + i))
 };
 }

 handleClick(letter) {
 this.setState({ justClicked: letter });
 }

 render() {
 return (
 <>
 Just clicked: {this.state.justClicked}
  <ul>
 {this.state.letters.map((letter) => (
 <li key={letter} onClick={() => this.handleClick(letter)}>
 {letter}
 </li>
 ))}
 </ul>
 </>
 );
 }
}
```

☆

↥

Q. When should we use arrow functions with React?

Arrows prevent `this` bugs
Arrow functions don not redefine the value of `this` within their function body. This makes it a lot easier to predict their behavior when passed as callbacks, and prevents bugs caused by use of this within callbacks. Using inline arrow functions in function components is a good way to achieve some decoupling.
Example:

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

class Button extends React.Component {
 render() {
 return (
 <button onClick={this.handleClick} style={this.state}>
 Set background to red
 </button>
 )
 }

 handleClick = () => {
 this.setState({ backgroundColor: 'red' })
 }
}

ReactDOM.render(
 <Button />,
 document.getElementById('root')
)
```

1. When we use `this` it generates a new function on every render, which will obviously have a new reference. 2. If the component we pass this generated function to is extending `PureComponent()`, it will not be able to bail out on rerendering, even if the actual data has not changed.

↥

Q. Is it good to use arrow functions in render

methods? **Problem:**

The **bind()** method creates a new function that, when called, has its this keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called. When arrow functions and binds in render. It breaks performance optimizations like `shouldComponentUpdate()` and `PureComponent`.

Example:

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

 deleteUser = id => {
 this.setState(prevState => {
 return {
 users: prevState.users.filter(user => user.id !== id)
 };
 });
 };

 render() {
 return (
 <div>
 <ul>
 {this.state.users.map(user => {
 return (
 <User key={user.id} name={user.name} id={user.id}
 onDeleteClick={() => this.deleteUser(user.id)}
 />
 );
 })}
 </ul>
 </div>
 );
 }
}
```

✩

Solution:

In below example, **App Component** has no arrow function in render. Instead, the relevant data is passed down to **User Component**. In User Component, `onDeleteClick()` calls the onClick function passed in on props with the relevant user.id. `// User Component`

```
class User extends React.PureComponent {
 onDeleteClick = () => {
 // No bind needed since we can compose the relevant data for this item here
 this.props.onClick(this.props.user.id);
 };

 render() {
 console.log(`${this.props.user.name} just rendered`);
 return (
 <li>
 <input type="button" value="Delete" onClick={this.onDeleteClick} />
 {this.props.user.name}
 </li>
 );
 }
}

// App Component

class App extends React.Component {
 constructor(props) {
 super(props);
 ...
 }

 deleteUser = id => {
 this.setState(prevState => {
 return {
 users: prevState.users.filter(user => user.id !== id)
 };
 });
```

```
};
renderUser = user => {
return <User key={user.id} user={user} onClick={this.deleteUser} />;
}

render() {
return (
<div>
<ul>
{this.state.users.map(this.renderUser)}
</ul>
</div>
);
}
}
```

↥

Q. How to avoid the need for binding in React?

1. Use Arrow Function in Class Property:

Usually when we want to access this inside a class method we would need to bind it to method like so:

```
class Button extends Component {
 constructor(props) {
 super(props)
 this.state = { clicked: false }
 }
 handleClick = () => this.setState({ clicked: true })
 render() {
 return <button onClick={this.handleClick}>Click Me</button>
 }
}
```

Binding `this` to `handleClick()` in the `constructor()` allows us to use `this.setState()` from Component inside `handleClick()`. **2. Bind in Render:**

```
onChange={this.handleChange.bind(this)}
```

This approach is terse and clear, however, there are performance implications since the function is reallocated on every

render. **3. Bind in Constructor:**

One way to avoid binding in render is to bind in the constructor

```
constructor(props) {
 super(props)
 this.handleChange = this.handleChange.bind(this)
}
```

This is the approach currently recommended in the React docs for "better performance in your application". ↥

Q. How do I bind a function to a component instance?

There are several ways to make sure functions have access to component attributes like `this.props` and `this.state`, depending on which syntax and build steps you are using.

1. Bind in Constructor (ES5):

```
class App extends Component {
 constructor(props) {
 super(props)
 this.handleClick = this.handleClick.bind(this)
 }
 handleClick() {
 console.log('Click happened')
 }
 render() {
 return <button onClick={this.handleClick}>Click Me</button>
```

```
 }
}
```

2. Class Properties:

```
class App extends Component {
 // Note: this syntax is experimental and not standardized yet.
 handleClick = () => {
 console.log('Click happened')
 }
 render() {
 return <button onClick={this.handleClick}>Click Me</button>
 }
}
```

3. Bind in Render:

```
class App extends Component {
 handleClick() {
 console.log('Click happened')
 }
 render() {
 return <button onClick={this.handleClick.bind(this)}>Click Me</button>
 }
}
```

Note: Using `Function.prototype.bind` in render creates a new function each time the component renders, which may have performance implications

4. Arrow Function in Render:

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

Note: Using an arrow function in render creates a new function each time the component renders, which may break optimizations based on strict identity comparison.

↥

Q. How can I prevent a function from being called too quickly?

1. Throttle:

Throttling prevents a function from being called more than once in a given window of time.

2. Debounce:

Debouncing ensures that a function will not be executed until after a certain amount of time has passed since it was last called. This can be useful when you have to perform some expensive calculation in response to an event that might dispatch rapidly (eg scroll or keyboard events).

Example:

```
/**
 * Throttle and Debounce in React
 */
import * as React from "react";
import * as _ from "lodash";

export default class App extends React.Component {
 state = { count: 0 };
 handleCount() {
 this.setState((state) => ({
 count: state.count + 1
 }));
 }
```

```
 // You will run count() only once after 100ms
 handleDebounce = _.debounce(() => this.handleCount(), 100);

 // You will run count() every 200ms
 handleThrottle = _.throttle(() => this.handleCount(), 200);

 render() {
 return (
 <div>
 {this.state.count}
 <hr />
 <button onClick={this.handleThrottle}>Click Me - Throttle </button>
 <button onClick={this.handleDebounce}>Click Me - Debounce </button>
 </div>
 );
 }
}
```

☆

3. RequestAnimationFrame Throttling:

The **requestAnimationFrame** is a way of queuing a function to be executed in the browser at the optimal time for rendering performance. A function that is queued with requestAnimationFrame will fire in the next frame. The browser will work hard to ensure that there are 60 frames per second (60 fps). However, if the browser is unable to it will naturally limit the amount of frames in a second.

For example, a device might only be able to handle 30 fps and so you will only get 30 frames in that second. Using requestAnimationFrame for throttling is a useful technique in that it prevents you from doing more than 60 updates in a second. If you are doing 100 updates in a second this creates additional work for the browser that the user will not see anyway.

```
/**
 * RequestAnimationFrame Throttling
 */
import rafSchedule from "raf-schd";
import React from "react";

export default class App extends React.Component {
 constructor(props) {
 super(props);

 this.handleScroll = this.handleScroll.bind(this);

 // Create a new function to schedule updates.
 this.scheduleUpdate = rafSchedule((point) => this.props.onScroll(point));
 }

 handleScroll(e) {
 // When we receive a scroll event, schedule an update.
 // If we receive many updates within a frame, we'll only publish the latest value.
 this.scheduleUpdate({ x: e.clientX, y: e.clientY });
 }

 componentWillUnmount() {
 // Cancel any pending updates since we're unmounting.
 this.scheduleUpdate.cancel();
 }

 render() {
 return (
 <div style={{ overflow: "scroll" }} onScroll={this.handleScroll}>
 <img src="/my-huge-image.png" alt="Nature" />
 </div>
 );
 }
}
```

☆

↥

Q. Explain synthetic event in React js?

Inside React event handlers, the event object is wrapped in a `SyntheticEvent` object. These objects are pooled, which means that the objects received at an event handler will be reused for other events to increase performance. This also means that accessing the event object's properties asynchronously will be impossible since the event's properties have been reset due to reuse. The

following piece of code will log null because event has been reused inside the SyntheticEvent pool:

```
function handleClick(event) {
 setTimeout(function () {
 console.log(event.target.name)
 }, 1000)
}
```

To avoid this we need to store the event's property:

```
function handleClick(event) {
 let name = event.target.name
 setTimeout(function () {
 console.log(name)
 }, 1000)
}
```

SyntheticEvent Object

```
void preventDefault()
void stopPropagation()
boolean isPropagationStopped()
boolean isDefaultPrevented()
void persist()
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
DOMEventTarget target
number timeStamp
string type
```

↥

Q. What is Event Pooling in React?

The `SyntheticEvent` is pooled. This means that the SyntheticEvent object will be reused and all properties will be nullified after the event callback has been invoked. This is for performance reasons. As such, you cannot access the event in an `asynchronous` way. **Example:**

```
function onClick(event) {
 console.log(event) // => nullified object.
 console.log(event.type) // => "click"
 const eventType = event.type // => "click"

 setTimeout(function() {
 console.log(event.type) // => null
 console.log(eventType) // => "click"
 }, 0)

 // Won't work. this.state.clickEvent will only contain null values.
 this.setState({clickEvent: event})

 // You can still export event properties.
 this.setState({eventType: event.type})
}
```

If we want to access the event properties in an asynchronous way, we should call `event.persist()` on the event, which will remove the synthetic event from the pool and allow references to the event to be retained by user code.

↥

Q. How to trigger click event programmatically?

We can use `ref` prop to acquire a reference to the underlying `HTMLInputElement` object through a callback, store the reference as a class property, then use that reference to later trigger a click from your event handlers using the `HTMLElement.click` method. **Example:**

```
class MyComponent extends React.Component {

 render() {
 return (
 <div onClick={this.handleClick}>
```

```
 <input ref={input => this.inputElement = input} />
 </div>
 )
 }

 handleClick = (e) => {
 this.inputElement.click()
 }
}
```

Note: The `ES6 arrow function` provides the correct lexical scope for `this` in the callback.

↥

Q. How to listen for click events that are outside of a component?

Example:

```
class OutsideAlerter extends Component {
 // ...
 componentDidMount() {
 document.addEventListener("mousedown", this.handleClickOutside);
 }

 componentWillUnmount() {
 document.removeEventListener("mousedown", this.handleClickOutside);
 }

 /**
 * Set the wrapper ref
 */
 setWrapperRef(node) {
 this.wrapperRef = node;
 }

 /**
 * Alert if clicked on outside of element
 */
 handleClickOutside(event) {
 if (this.wrapperRef && !this.wrapperRef.contains(event.target)) {
 alert("You clicked outside of me!");
 }
 }

 render() {
 return <div ref={this.setWrapperRef}>{this.props.children}</div>;
 }
}

OutsideAlerter.propTypes = {
 children: PropTypes.element.isRequired
};
```

☆

↥

Q. How to convert text to uppercase on user input entered?

```
import React, { useState } from "react"
import ReactDOM from "react-dom"

const toInputUppercase = e => {
 e.target.value = ("" + e.target.value).toUpperCase()
}

const App = () => {
 const [name, setName] = useState("")

 return (
 <input
 name={name}
```

```
 onChange={e => setName(e.target.value)}
 onInput={toInputUppercase} // apply on input which do you want to be capitalize
 />
 )
}

ReactDOM.render(<App />, document.getElementById("root"))
```

↥

Q. How to set a dynamic key for state?

1. Dynamic Key:

```
onChange(e) {
 const key = e.target.name
 const value = e.target.value
 this.setState({ [key]: value })
}
```

2. Nested States:

```
handleSetState(cat, key, val) {
 const category = {...this.state[cat]}
 category[key] = val
 this.setState({ [cat]: category })
}
```

↥

Q. What are the pointer events in React?

Pointer events, in essence, are very similar to mouse events (mousedown, mouseup, etc.) but are hardware-agnostic and thus can handle all input devices such as a mouse, stylus or touch. This is great since it removes the need for separate implementations for each device and makes authoring for cross-device pointers easier.

The API of pointer events works in the same manner as existing various event handlers. Pointer events are added as attributes to React component and are passed a callback that accepts an event. Inside the callback we process the event.

The following event types are now available in React DOM

- onPointerDown
- onPointerMove
- onPointerUp
- onPointerCancel
- onGotPointerCapture
- onLostPointerCapture
- onPointerEnter
- onPointerLeave
- onPointerOver
- onPointerOut

Example: Drag and Drop using Point Events

```
// App Component
import React, { Component } from 'react'
import logo from './logo.svg'
import './App.css'
import DragItem from './DragItem'

class App extends Component {
 render() {
 return (
 <div className="App">
```

```
 <header className="App-header">
 <img src={logo} className="App-logo" alt="logo" />  <h1
className="App-title">Welcome to React sample of Point Events</h1>  </header>
 <div className="App-intro">
 <DragItem />
 </div>
 </div>
 )
 }
}
export default App
```

DragItem Component

```
import React from 'react'
const CIRCLE_DIAMETER = 100

export default class DragItem extends React.Component {

 state = {
 gotCapture: false,
 circleLeft: 500,
 circleTop: 100
 }
 isDragging = false
 previousLeft = 0
 previousTop = 0

 onDown = e => {
 this.isDragging = true
 e.target.setPointerCapture(e.pointerId)
 this.getDelta(e)
 }
 onMove = e => {
 if (!this.isDragging) {
 return
 }

 const {left, top} = this.getDelta(e)
 this.setState(({circleLeft, circleTop}) => ({
 circleLeft: circleLeft + left,
 circleTop: circleTop + top
 }))
 }
 onUp = e => (this.isDragging = false)
 onGotCapture = e => this.setState({gotCapture: true})
 onLostCapture = e => this.setState({gotCapture: false})
 getDelta = e => {
 const left = e.pageX
 const top = e.pageY
 const delta = {
 left: left - this.previousLeft,
 top: top - this.previousTop,
 }
 this.previousLeft = left
```