**What are React Hooks?**

****Hooks**** are a new addition in React 16.8. They let you use state and other React features without writing a class. With Hooks, you can extract stateful logic from a component so it can be tested independently and reused. Hooks allow you to reuse stateful logic without changing your component hierarchy. This makes it easy to share Hooks among many components or with the community.

**How to access DOM elements in React?**

One of the useful application of the useRef() hook is to access DOM elements. This is performed in 3 steps:

1. Define the reference to access the element const elementRef = useRef();
2. Assign the reference to ref attribute of the element: <div ref={elementRef}></div>;
3. After mounting, elementRef.current points to the DOM element.

Consider:

import { useRef, useEffect } from 'react';

function AccessingElement() {

const elementRef = useRef();

useEffect(() => {

const divElement = elementRef.current;

console.log(divElement); // logs <div>I'm an element</div>

}, []);

return (

<div ref={elementRef}>

I'm an element

</div>

);}

**How to call loading function with React useEffect only once?**

If you only want to run the function given to useEffect after the initial render, you can give it an empty array [] as the second argument.

For example:

function MyComponent(){

useEffect(() => {

loadDataOnlyOnce();

}, []);

return <div> { /\*...\*/} </div>;}

**Provide an example of any simple Custom React Hook. Why do we need Custom Hooks?**

A ****Custom**** Hook is a ****stateful function**** that uses other react built-in hooks (e.g. useState, useCallback etc.) that can wrap around the stateful logic that you wanted to gather in one place and avoid copying and pasting the same logic in multiple components.

Consider the increment/decriment custom hook:

const useCounter = () => {

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

return {

counter, // counter value

increment: () => setCounter(counter + 1), // function 1

decrement: () => setCounter(counter - 1) // function 2

};};

and then in your component you can use it as follows:

const Component = () => {

const { counter, increment, decrement } = useCounter();

return (

<div>

<span onClick={decrement}>-</span>

<span style={{ padding: "10px" }}>{counter}</span>

<span onClick={increment}>+</span>

</div>

);}

**What is useState() in React?**

**Junior**

**Problem**

Explain what is the use of useState(0) there:

...const [count, setCounter] = useState(0);const [moreStuff, setMoreStuff] = useState(...);...

const setCount = () => {

setCounter(count + 1);

setMoreStuff(...);

...};

**Answer**

useState is one of build-in react hooks. useState(0) returns a tuple where the first parameter count is the current state of the counter and setCounter is the method that will allow us to update the counter's state.

We can use the setCounter method to update the state of count anywhere - In this case we are using it inside of the setCount function where we can do more things; the idea with hooks is that we are able to keep our code more functional and avoid class based components if not desired/needed.

**Can you initialise state from a function? Provide and example**

****Yes****! Consider:

const StateFromFn = () => {

const [token] = useState(() => {

let token = window.localStorage.getItem("my-token");

return token || "default#-token#"

})

return <div>Token is {token}</div>}

**Do two components using the same Hook share state?**

**Answer**

No. Custom Hooks are a mechanism to reuse stateful logic (such as setting up a subscription and remembering the current value), but every time you use a custom Hook, all state and effects inside of it are fully isolated.

**Explain the difference between useState and useRef hooks?**

**Mid**

**Answer**

1. Updating a reference created by useRef doesn't trigger re-rendering, while updating the state (setState) makes the component re-render;
2. useRef returns an object with a current property holding the actual value. In contrast, useState returns an array with two elements.
3. useRef‘s current property is mutable, but useState‘s state variable is not.
4. The reference update is synchronous (the updated reference value is available right away), while the state update is asynchronous (the state variable is updated after re-rendering).

Using useRef - no re-renders

const countRef = useRef(0);

const handle = () => {

countRef.current++;

console.log(`Clicked ${countRef.current} times`);};

Using useState - triggers re-render

const [count, setCount] = useState(0);

const handle = () => {

const updatedCount = count + 1;

console.log(`Clicked ${updatedCount} times`);

setCount(updatedCount);};

**How can I make use of Error Boundaries in functional React components?**

**Mid**

**Answer**

As of v16.2.0, there's no way to turn a functional component into an error boundary. The componentDidCatch() method works like a JavaScript catch {} block, but for components. ****Only class components can be error boundaries. In practice, most of the time you’ll want to declare an error boundary component once and use it throughout your application.****

Also bear in mind that try/catch blocks won't work on all cases. If a component deep in the hierarchy tries to update and fails, the try/catch block in one of the parents won't work -- because it isn't necessarily updating together with the child.

A few third party packages on npm implement error boundary hooks.

**How to use componentWillMount() in React Hooks?**

You cannot use any of the existing lifecycle methods (componentDidMount, componentDidUpdate, componentWillUnmount etc.) in a hook. They can only be used in class components. And with Hooks you can only use in functional components.

You can think of useEffect Hook as

componentDidMount, componentDidUpdate, and componentWillUnmount combined.

Code inside **componentDidMount** run once when the component is mounted. useEffect hook equivalent for this behaviour is

useEffect(() => {

// Your code here}, []);

Without the second parameter the useEffect hook will be called on every render (like componentDidUpdate) of the component which can be dangerous:

useEffect(() => {

// Your code here});

Hook equivalent of componentWillUnmount() code will be as follows

useEffect(() => {

window.addEventListener('mousemove', () => {});

// returned function will be called on component unmount

return () => {

window.removeEventListener('mousemove', () => {})

}}, [])

**What are common use cases for the useMemo?**

**Answer**

The primary purpose of useMemo hook is "performance optimization".

* It returns a memoized value,
* It accepts two arguments - ****create**** function (which should return a value to be memoized) and ****dependency**** array. It will recompute the memoized value only when one of the dependencies has changed.

Using useMemo you achieve:

* referential equality of the values (to further send them to props of the components to potentially avoid re-renders)
* eliminate redoing of the computationally expensive operations for same parameters

For example:

function App() {

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

function format() {

console.log('formatting....'); // this will print only when data has changed

const formattedData = [];

data.forEach(item => {

const newItem = //...do soemthing here,

if (newItem) {

formattedData.push(newItem);

}

})

return formattedData;

}

const formattedData = useMemo(format, [data])

return (

<>

{formattedData.map(item => (

<div key={item.id}>

{item.title}

</div>

))}

</>

) }

**What are differences between React.memo() and useMemo()?**

**Answer**

* React.memo() is a higher-order component (HOC) that we can use to wrap ****components**** that we do not want to re-render unless props within them change
* useMemo() is a React Hook that we can use to wrap ****functions**** within a component. We can use this to ensure that the values within that function are re-computed only when one of its dependencies change

**What are production use cases for the useRef?**

**Answer**

* useRef simply returns a plain Javascript object. Its value can be accessed and modified (mutability) as many times as you need without worrying about rerender.
* useRef value will persist (won't be reset to the initialValue unlike an ordinary object defined in your function component; it persists because useRef gives you the same object instead of creating a new one on subsequent renders) for the component lifetime and across re-renders.
* useRef hook is often used to store values instead of DOM references. These values can either be a state that does not need to change too often or a state that should change as frequently as possible but should not trigger full re-rendering of the component.

For example:

const refObject = useRef(initialValue);

**What is equivalent of this code using React Hooks?**

**Problem**

Let's say in our project we have componentWillUnmount that is used for cleanup (like removing event listeners, cancel the timer etc). How to refactor this code using React Hooks?

componentDidMount() {

window.addEventListener('mousemove', () => {})}

componentWillUnmount() {

window.removeEventListener('mousemove', () => {})}

**Answer**

React Hooks equivalent of above code will be as follows

useEffect(() => {

window.addEventListener('mousemove', () => {});

// returned function will be called on component unmount

return () => {

window.removeEventListener('mousemove', () => {})

}}, [])

**When would you use useContext hook?**

React’s useContext hook makes it easy to ****pass data throughout your app**** without manually passing props down the tree. React Context is a way to ****manage state globally****.

Consider:

import { useState, createContext } from "react";import ReactDOM from "react-dom/client";

const UserContext = createContext()

Wrap child components in the Context Provider and supply the state value.

function Component1() {

const [user, setUser] = useState("Jesse Hall");

return (

<UserContext.Provider value={user}>

<h1>{`Hello ${user}!`}</h1>

<Component2 user={user} />

</UserContext.Provider>

);}

Then you can access the user Context in all components:

import { useState, createContext, useContext } from "react";

function Component5() {

const user = useContext(UserContext);

return (

<>

<h1>Component 5</h1>

<h2>{`Hello ${user} again!`}</h2>

</>

);}

**When would you use useRef?**

**Answer**

The main use cases:

To store a ref to ****DOM**** elements so you can later do something with them:

Consider:

function TextInputWithFocusButton() {

const inputEl = useRef(null);

const onButtonClick = () => {

inputEl.current.focus();

};

return (

<>

<input ref={inputEl} type="text"/>

<button onClick={onButtonClick}>Focus the input</button>

</>

);}

To store values without triggering re-renders:

Consider:

function Counter(){

const [count, setCount] = useState(0);

const prevCountRef = useRef();

useEffect(() => {

prevCountRef.current = count;

});

const prevCount = prevCountRef.current;

return <h1>Now: {count}, before: {prevCount} </h1>;}

**When writing a Custom Hook, what is the difference between it and a normal function?**

**Mid**

**Answer**

Hooks use a stateful closure around the invocation of the function component to store the state on behalf of that component instance. That closure is maintained by React.

* Custom hook will only be "stateful" if you use state with useState inside (or something that implements useState),
* Hooks should be called from the React code only not from the regular JS functions. Hence, Hooks' scope is limited to the React code world and has more power to do a lot with React code,
* In the class-based components, the Hooks won't work but regular functions will.
* In the regular JS functions, you can't access useState, useEffect, useContext etc. but in react custom hooks I can.

**Which lifecycle methods of class component is replaced by useEffect in functional component?**

The lifecyce methods replaced by useEffect Hooks of functional component are componentDidMount(), componentDidUpdate(), and componentWillUnmount()

componentDidMount: is equivalent for running an effect once.

For example:

useEffect(() => {

console.log("This is useEffect Hook equivalent of componentDidMount lifecycle method")},[]);

****Note:**** empty array = useEffect hook runs once on mount

componentDidUpdate: is equivalent for running effects when things change

For example

useEffect(() => {

console.log("The name props has changed!");}, [props.name]);

componentWillUnmount: To run a hook as the component is about to unmount, we just have to return a function from the useEffect Hook

For example:

useEffect(() => {

console.log('running effect');

return () => {

console.log('unmount');

}})

**Do Hooks replace render props and higher-order components (HOC)?**

**How do I update state on a nested object with useState()?**

**How to mitigate multiple component re-renders when using multiple useState calls?**

**Provide a good example of using useCallback hook in React**

**What's the difference between useCallback and useMemo in practice?**

**When would you use flushSync in ReactJS?**

**Can a custom React hook return JSX?**

**Explain the difference between useMemo vs useEffect + useState usage in that code**

**How would you optimise this code using one of the React Hooks?**