React Class Components

- Class components are a type of component in React that are defined as JavaScript classes.
- They extend the `React.Component` base class and have access to the React class component life cycle methods.
- They can accept props as an argument to their constructor or via the `this.props` object.
- They can have state defined via the `this.state` object and can update state via the `this.setState()` method.
- They can render a React element using the `render()` method, which returns a single React element.
- They can define methods that can be used for event handling, data manipulation, or other logic.
- They can have access to the `componentDidMount()`, `componentDidUpdate()`, `componentWillUnmount()`, and other life cycle methods, which allow you to run code at different points in the component's lifecycle.
- They can also use `shouldComponentUpdate()` to optimize rendering and avoid unnecessary updates.
- They can be more verbose and require more boilerplate code than functional components, but can be useful for more complex components or when working with third-party libraries that require class components.
- They are gradually being replaced by functional components and hooks, which are simpler and more lightweight.

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

```
class CounterClass extends Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }

  handleIncrement = () => {
    this.setState((prevState) => ({ count: prevState.count + 1 }));
  };

  handleDecrement = () => {
    this.setState((prevState) => ({ count: prevState.count - 1 }));
  };

  render() {
    return (
      <>
        <h1>Counter</h1>
        <button onClick={this.handleDecrement}>-</button>
        <span>{this.state.count}</span>
        <button onClick={this.handleIncrement}>+</button>
      </>
    );
  }
}

export default CounterClass;
```

React Functional Components and Hooks

React Functional Components

- Functional components are a type of component in React that are defined as JavaScript functions instead of classes.
- They accept props as the first argument and return a React element to be rendered.
- They are simpler and more lightweight than class components, and are recommended for most use cases.
- They can also be easier to reason about and test, since they don't have the added complexity of the class component life cycle methods.

React Hooks

- React Hooks are a set of functions that allow functional components to use state and other React features that were previously only available to class components.
- They were introduced in React 16.8 as a way to simplify state management and make functional components more powerful.
- The most commonly used hooks are useState, useEffect, useContext, and useRef, but there are several others as well.
- useState allows functional components to have state variables that can be updated and used in rendering.
- useEffect allows functional components to perform side effects, such as fetching data, subscribing to events, or updating the DOM.
- useContext allows functional components to access a context object, which can be used to share data between components without the need for props drilling.
- useRef allows functional components to store a mutable value that persists across renders, which can be useful for storing references to DOM elements or other mutable values.

Overall, functional components and React Hooks have made it easier to write clean and maintainable code in React, and have helped to simplify state management and reduce boilerplate code.

useState

- useState is a Hook that lets you add React state to function components.
- useState returns a pair: the current state value and a function that lets you update it.
- useState is similar to this.state in a class, except it doesn’t merge the old and new state together.
- useState is a new way to use the exact same capabilities that this.state provides in a class.
- We call it inside a function component to add some local state to it.
- React will preserve this state between re-renders.
- It must not be called conditionally because that would prevent React from tracking it.
- It also must not be called nested in another function because that would make it a local variable that would be re-initialized on every render.
- Hooks don’t work inside classes. But you can use them instead of writing classes.

- Hooks are functions that let you “hook into” React state and lifecycle features from function components.

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

function CounterFunction() {
  const [count, setCount] = useState(() => 0);

  const handleIncrement = () => {
    setCount((prevCount) => prevCount + 1);
  };

  const handleDecrement = () => {
    setCount((prevCount) => prevCount - 1);
  };

  return (
    <>
      <h1>Counter</h1>
      <button onClick={handleDecrement}>-</button>
      <span>{count}</span>
      <button onClick={handleIncrement}>+</button>
    </>
  );
}

export default CounterFunction;
```

useEffect

`useEffect` is a built-in React hook that allows you to add side effects to your functional components. Here are some key points about `useEffect`

- `useEffect` is called after every render of the component.
- `useEffect` takes two arguments: a function that will be executed after each render, and an optional array of dependencies.
- If the dependency array is not provided, the effect will be called after every render.
- If the dependency array is provided, the effect will only be called when the values in the array change.
- The function passed to `useEffect` can return a cleanup function that will be called before the next render or before the component unmounts.
- `useEffect` can be used to perform various side effects, such as fetching data, updating the DOM, or subscribing to events.

- `useEffect` can be used to replicate the behavior of various class component life cycle methods, such as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.
- The order of multiple `useEffect` calls within a component matters. Effects without dependencies are executed in the order they are declared, and effects with dependencies are executed after the effects without dependencies.

ComponentDidMount

```
const [windowWidth, setWindowWidth] = useState(window.innerWidth);

const handleResize = () => {
  setWindowWidth(window.innerWidth);
};

useEffect(() => {
  console.log("onMount");
  window.addEventListener("resize", handleResize);

  // clean up function
  return () => {
    window.removeEventListener("resize", handleResize);
  };
}, []);
```

ComponentDidUpdate

```
const [resourceType, setResourceType] = useState(() => "posts");

useEffect(() => {
  console.log("onUpdate");
}, [resourceType]);
```

ComponentWillUnmount

```
useEffect(() => {
  return () => {
    console.log("onUnmount");
  };
}, []);
```

Social App to understand useEffect

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

export default function Social() {
  const [resourceType, setResourceType] = useState(() => "posts");
  const [items, setItems] = useState([]);

  const [windowWidth, setWindowWidth] = useState(window.innerWidth);

  const handleResize = () => {
    setWindowWidth(window.innerWidth);
  };

  useEffect(() => {
    console.log("onMount");
    window.addEventListener("resize", handleResize);

    // clean up function
    return () => {
      window.removeEventListener("resize", handleResize);
    };
  }, []);

  useEffect(() => {
    console.log("onUpdate");
    fetch(`https://jsonplaceholder.typicode.com/${resourceType}`)
      .then((response) => response.json())
      .then((json) => setItems(json));
  }, [resourceType]);

  useEffect(() => {
    return () => {
      console.log("onUnmount");
    };
  }, []);

  return (
    <>
      <div>
        <button onClick={() => setResourceType("posts")}>Posts</button>
        <button onClick={() => setResourceType("users")}>Users</button>
        <button onClick={() => setResourceType("comments")}>Comments</button>
      </div>
      <h1>Window Width: {windowWidth}</h1>
      <h1>{resourceType}</h1>
      {resourceType === "posts" && <Posts items={items} />}
      {resourceType === "users" && <Users items={items} />}
      {resourceType === "comments" && <Comments items={items} />}
      {items.map((item) => {
        return <pre key={item.id}>{JSON.stringify(item)}</pre>;
      })}
    </>
  );
}

function Users({ items }) {
  return (
    <>
      {items.map((item) => {
        const { id, name, username, email, address, phone, website, company } =
          item;
        const { street, suite, city, zipcode, geo } = address;
        const { lat, lng } = geo;

        return (
          <div key={id}>
            <h2>{name}</h2>
            <p>Username: {username}</p>
            <p>Email: {email}</p>
            <p>
              Address: {street}, {suite}, {city}, {zipcode}
            </p>
            <p>
              Latitude: {lat}, Longitude: {lng}
            </p>
            <p>Phone: {phone}</p>
            <p>Website: {website}</p>
            <p>Company: {company.name}</p>
            <p>Catchphrase: {company.catchPhrase}</p>
            <p>Business: {company.bs}</p>
          </div>
        );
      })}
    </>
  );
}

function Posts({ items }) {
  return (
    <>
      {items.map((item) => {
        const { id, title, body } = item;

        return (
          <div key={id}>
            <h2>{title}</h2>
            <p>{body}</p>
          </div>
        );
      })}
    </>
  );
}

function Comments({ items }) {
  return (
    <>
      {items.map((item) => {
        const { id, name, email, body } = item;

        return (
          <div key={id}>
            <h2>{name}</h2>
            <p>{email}</p>
            <p>{body}</p>
          </div>
        );
      })}
    </>
  );
}
```

this binding in class components

In React class components, there are several ways to handle method binding. Here are four common approaches:

1. Binding in the constructor:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // Handle click event
  }

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

In this approach, you explicitly bind the method within the component's constructor using the `bind` method. It ensures that `this` refers to the component instance when the method is called.

2. Arrow function class property:

```
class MyComponent extends React.Component {
  handleClick = () => {
    // Handle click event
  };

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

By using an arrow function for the method, you automatically bind it to the component instance. This approach eliminates the need for explicit binding in the constructor.

3. Binding in the event handler:

```
class MyComponent extends React.Component {
  handleClick() {
    // Handle click event
  }

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

You can also bind the method directly in the event handler within the render method. It ensures that the method is bound correctly when the event occurs.

4. Binding with `Function.prototype.bind()` in JSX:

```
class MyComponent extends React.Component {
  handleClick() {
    // Handle click event
  }

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

Similar to the previous approach, you can use the `bind()` method directly within the JSX to bind the method to the component instance.

All of these approaches achieve method binding in React class components, but the choice depends on your preference and coding style. It's important to ensure that `this` is correctly bound to the component instance when handling events or passing methods as callbacks.