**Understanding the Hooks**

**React Hooks**

Hooks were added to React in version 16.8.

Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed.

Although Hooks generally replace class components, there are no plans to remove classes from React.

**What is Hooks**

Hooks allow us to "hook" into React features such as state and lifecycle methods.

You must import Hooks from react.

Here we are using the useState Hook to keep track of the application state.

State generally refers to application data or properties that need to be tracked.

**Hook Rules**

There are 3 rules for hooks:

* Hooks can only be called inside React function components.
* Hooks can only be called at the top level of a component.
* Hooks cannot be conditional

**React useState Hook**

The React useState Hook allows us to track state in a function component.

State generally refers to data or properites that need to be tracking in an application.

**Import useState**

To use the useState Hook, we first need to import it into our component.

**Example:**

At the top of your component, import the useState Hook.

import { useState } from "react";

**Initialize useState**

We initialize our state by calling useState in our function component.

useState accepts an initial state and returns two values:

* The current state.
* A function that updates the state.

**Example:**

Initialize state at the top of the function component.

import { useState } from "react";

function FavoriteColor() {

const [color, setColor] = useState("");

}

Notice that again, we are destructuring the returned values from useState.

* The first value, color, is our current state.
* The second value, setColor, is the function that is used to update our state.

**Read State**

We can now include our state anywhere in our component.

**Example:**

Use the state variable in the rendered component.

import { useState } from "react";

import ReactDOM from "react-dom";

function FavoriteColor() {

const [color, setColor] = useState("red");

return <h1>My favorite color is {color}!</h1>

}

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

**Update State**

To update our state, we use our state updater function.

We should never directly update state. Ex: color = "red" is not allowed.

**Example:**

Use a button to update the state:

import { useState } from "react";

import ReactDOM from "react-dom";

function FavoriteColor() {

const [color, setColor] = useState("red");

return (

<>

<h1>My favorite color is {color}!</h1>

<button

type="button"

onClick={() => setColor("blue")}

>Blue</button>

</>

)

}

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

**Updating Objects and Arrays in State**

* When state is updated, the entire state gets overwritten.
* What if we only want to update the color of our car?
* If we only called setCar({color: "blue"}), this would remove the brand, model, and year from our state.
* We can use the JavaScript spread operator to help us.

Example:

Use the JavaScript spread operator to update only the color of the car:

import { useState } from "react";

import ReactDOM from "react-dom";

function Car() {

const [car, setCar] = useState({

brand: "Ford",

model: "Mustang",

year: "1964",

color: "red"

});

const updateColor = () => {

setCar(previousState => {

return { ...previousState, color: "blue" }

});

}

return (

<>

<h1>My {car.brand}</h1>

<p>

It is a {car.color} {car.model} from {car.year}.

</p>

<button

type="button"

onClick={updateColor}

>Blue</button>

</>

)

}

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

**React useEffect Hooks**

The useEffect Hook allows you to perform side effects in your components.

Some examples of side effects are: fetching data, directly updating the DOM, and timers.

useEffect accepts two arguments. The second argument is optional.

useEffect(<function>, <dependency>)

**Let's use a timer as an example.**

useEffect runs on every render. That means that when the count changes, a render happens, which then triggers another effect.

This is not what we want. There are several ways to control when side effects run.

We should always include the second parameter which accepts an array. We can optionally pass dependencies to useEffect in this array.

**1. No dependency passed:**

useEffect(() => {

//Runs on every render

});

**2.** **An empty array:**

useEffect(() => {

//Runs only on the first render

}, []);

**3. Props or state values:**

useEffect(() => {

//Runs on the first render

//And any time any dependency value changes

}, [prop, state]);

So, to fix this issue, let's only run this effect on the initial render.

**Example:**

**Only run the effect on the initial render:**

import { useState, useEffect } from "react";

import ReactDOM from "react-dom";

function Timer() {

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

useEffect(() => {

setTimeout(() => {

setCount((count) => count + 1);

}, 1000);

} []); // <- add empty brackets here

return <h1>I've rendered {count} times!</h1>;

}

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

**Example:**

Here is an example of a useEffect Hook that is dependent on a variable. If the count variable updates, the effect will run again:

import { useState, useEffect } from "react";

import ReactDOM from "react-dom";

function Counter() {

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

const [calculation, setCalculation] = useState(0);

useEffect(() => {

setCalculation(() => count \* 2);

}, [count]); // <- add the count variable here

return (

<>

<p>Count: {count}</p>

<button onClick={() => setCount((c) => c + 1)}>+</button>

<p>Calculation: {calculation}</p>

</>

);

}

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

**Effect Cleanup**

Some effects require cleanup to reduce memory leaks.

Timeouts, subscriptions, event listeners, and other effects that are no longer needed should be disposed.

We do this by including a return function at the end of the useEffect Hook.

**Example:**

**Clean up the timer at the end of the useEffect Hook:**

import { useState, useEffect } from "react";

import ReactDOM from "react-dom";

function Timer() {

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

useEffect(() => {

let timer = setTimeout(() => {

setCount((count) => count + 1);

}, 1000);

return () => clearTimeout(timer)

}, []);

return <h1>I've rendered {count} times!</h1>;

}

ReactDOM.render(<Timer />, document.getElementById("root"));

**React useContext Hook**

**React Context**

React Context is a way to manage state globally.

It can be used together with the useState Hook to share state between deeply nested components more easily than with useState alone.

**The Solution**

The solution is to create context.

**Create Context**

To create context, you must Import createContext and initialize it:

import { useState, createContext } from "react";

import ReactDOM from "react-dom";

const UserContext = createContext()

Next we'll use the Context Provider to wrap the tree of components that need the state Context.

**Context Provider**

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>

);

}

Now, all components in this tree will have access to the user Context.

**Use the useContext Hook**

In order to use the Context in a child component, we need to access it using the useContext Hook.

First, include the useContext in the import statement:

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

Then you can access the user Context in all components:

function Component5() {

const user = useContext(UserContext);

return (

<>

<h1>Component 5</h1>

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

</>

);

}