useEffect Hook

As mentioned previously, the useEffect hook allows you to take advantage of certain things in the lifecycle in a much cleaner and more concise way. One of the nice things about the `useEffect` hook is that it allows you to combine three lifecycle hooks into a single function. Those functions are `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.

To review:

`componentDidMount` is a method called after the component is created and rendered into the DOM.

`componentDidUpdate` is called after the component has been updated and re-rendered, usually because the component's state or props have changed.

`componentWillUnmount` is the phase when a component is removed from the DOM.

With The useEffect Hook in Functional Components

With functional components, you don't have access to the lifecycle functions like `componentDidMount` etc. built into classes, but you do have the ability to use a hook. You can do so by importing the hook and implementing it in the component:

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

function Example() {
  // Hook to replace class based state
  const [count, setCount] = useState(0);

  // This replaces both the didMount AND the didUpdate and will set the title when
any of those things happen (essentially this will run after each render)
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount((count) => count + 1)}>Click me</button>
    </div>
  );
}
```

The `useEffect` hook requires a function that will run. This is your effect. Depending on how you set it up, it will run on mount, update, AND unmount. Something important to note is that it will ALWAYS run on mount.

Code Cleanup

One of the problems above is that there is no way (as the code is currently written) to stop some sort of process when a component un-mounts. While the example above doesn't actually require it, some functionality might. Let's take a look at an example like that and how you would approach it. Let's assume you have some sort of component that keeps track of friend status and want to stop paying attention to their status when you leave that component.

One of the nice things about the `useEffect` hook is that it comes prebuilt with the ability to stop the effect already. This is actually the return value of the effect function itself. If you return a function from the `useEffect`, that function will trigger on unmount automatically. Without a return, no extra functionality will run on unmount It would look like:

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  }, [count])

  // This will subscribe to a friend's online status when the component mounts
  // When the component unmounts, it unsubscribed or when the friend's 'id'
changes
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  }, [props.friend.id]);
}
```

If you notice above, there is now a return value in the `useEffect`. There are also now two useEffects. When you want something to happen when the component unmounts, you simply need to return a function from your `useEffect`. This function will automatically be run when the component un-mounts. In this case it unsubscribes from the friend status. One other important thing to note is that you should (as above) ideally have one `useEffect` PER effect / piece it impacts. Since we have one effect to update the title and one to update friend status, they live in separate effects. This also becomes important to stop effects from happening.

Props and Stopping Unneeded Updates

One problem you can run into is you can have an effect run even if there wasn't a change. Because the hook will try to run after each render, the title will be updated (even if it updates from 0 to 0) each time the friend status changes and vice versa. You can get around this in both classes and functions but it is MUCH easier with Hooks. One other problem you run into is handling lifecycle changes if the props passed in change.

With functional components and the `useEffect` Hook, it is very easy to keep some effect from triggering on EVERY didUpdate. The `useEffect` Hook can take a second argument which is simply the value(s) to watch. If that value changes, it will run the hook. If it doesn't, it won't:

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // Only re-run the effect if count changes
```

```
useEffect(() => {}, [value])
```

With the above, this makes it so the effect will run on mount (cannot stop that from happening) and then only when `count` changes. Make sure that whatever values (and you can have any you want) are in that dependency array are in the React ecosystem so React can see if they change. If you want something to ONLY happen on mount and not on any subsequent update, you can leave the array empty like:

```
useEffect(() => {
  // some effect
}, []);
```

```
const [username, setUsername] = useState("");
useEffect(() => {
  console.log(username);
}, [username]);
```

One way to think of it is that the effect will run on mount and then whenever `undefined` changes from being `undefined` (which will never happen).

Main things to remember. If you keep these in mind, useEffect will be easy to use:

- The code inside the hook will run when one of the dependencies has changed ([value1, value2, ...]).
- If the dependency array is empty ('[]'), the effect will only run once, when the component mounts.
- If there is no dependency array, the effect will run after every render.