# useEffect hook

- We know `useState`, `useId`, `useRef` hooks
  - `useEffect` is another

`useEffect()` used to create a **side-effect** of rendering

- `useEffect()` is passed a callback
- callback runs *after* the component renders

# Basic example

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

function App() {
  useEffect( () => console.log('in effect') );
  console.log('in app');
  return (
    <div className="app">
    </div>
  );
}
```

```
in app
in effect
```

# useEffect callback called on every rerender

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

function App() {
  const [ count, setCount ] = useState(0);
  useEffect( () => console.log('in effect') );
  console.log('in app');
  return (
    <div className="app">
      <button onClick={ () => setCount(count+1) }>
        {count}
      </button>
    </div>
  );
}
```

Each `in app` followed by an `in effect`

# Why is Console Showing Messages Twice?

React 18 added a feature

- In "development mode"
  - The dev server via `npm run dev`
- Components rendered a second time
  - Largely to highlight effect problems

Mostly you can ignore this

- Won't happen in production
  - The built files using `npm run build`
- But watch for surprises!

# Why "Effect"?

`useState` gives us a state

What does `useEffect` give us?

- A "side effect" of rendering
- "side effects" are something to minimize
- but can be useful

# useEffect dependency array

useEffect callback doesn't have to run on ALL renders

- Can be passed a second argument
- The **dependency array**
- Lists values to watch
- A change in a value triggers callback to run
    - Only checked on render though

# Dependency Array Demonstration

```
function App() {
  const [ count, setCount ] = useState(0);
  const [ watched, setWatched ] = useState(0);
  useEffect(
    () => console.log('in effect'),
    [ watched ],
  );
  console.log('in app');
  return (
    <div className="app">
      <button onClick={ () => setCount(count+1) }>
        Unwatched: {count}
      </button>
      <button onClick={ () => setWatched(watched+1) }>
        Watched: { watched }
      </button>
    </div>
  );
}
```

# Simple Results

- Whenever the `watched` value changed
    - useEffect callback was called
- When an unwatched value changed
    - useEffect callback NOT called

# Infinite Loop

If you change a state that is in the dependency array

```
const [state, setState] = useState(0);
useEffect(
  () => setState(state+1),
  [state, setState],
);
```

- Infinite Loop!

Either the useEffect callback

- should NOT change state it depends on
- OR it only conditionally changes the state

# What if empty deps array?

What if:

```
useEffect(
  () => console.log('in effect'),
  [],
);
```

# Empty dependency array results

- useEffect callback runs on first render
    - Not on any later renders
- If component is removed from page and reapplied
    - callback once again runs on first render
- If multiple instances of component
    - callback runs on first render of each instance

# When to use dependency array

First questions:

- What is your "effect"?
- Why are you doing so based on render?

Component will re-render each time state changes

- Do you want your effect each time state changes?

If your effect is based on 1+ values

- Those values are your dependency array

# Effect: Increasing Counter

Let's write a component that will show a Counter

- When the component FIRST renders, counter starts
- Automatically increments (roughly 1/second)
- Cleans up when component removed

Creating the increment is an "effect"

# Component Base Structure

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

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

  return (
    <div className="counter">{count}</div>
  );
}

export default Counter;
```

# Increase `count` ~1/second

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

useEffect(
  () => {
    setInterval( () => {
      console.log('incrementing');
      setCount(count + 1);
    }, 1000);
  }
);
```

But this has a problem!

# Too Many Effects

The interval was changing state (using `setCount()`)

- Which triggers a rerender
- Each render added a NEW effect

Easier to see with `setCount( count => count + 1 );`

We only want to create the interval once

- Use a dependency array

# Adding the dependency array

```
useEffect(
  () => {
    setInterval( () => {
      console.log('incrementing');
      setCount(count => count + 1);
    }, 1000);
  },
  [] // empty = effect on first render only
);
```

We DO NOT want `count` as a dependency

- It changes = infinite loop
- Using the function form for setter works fine
- Leaving `count` out will generate a warning
    - Unless we use pass a function to the setter

# Why is counter going up by 2?

This is because of that development feature

- Our effect is running twice

Why would they mess us up like this?

- Actually a sign of a problem in our code
- Let's look at that problem first
    - Then consider why double render helped

# We still have a problem

`<Counter>` works fine

- Double count not withstanding
- As long as it is on the page
- What happens when removed?

```
function App() {
  const [showCounter, setShowCounter] = useState(false);
  return (
    <div className="app">
      <button
        onClick={ () => { setShowCounter( !showCounter ) } }
      >Toggle Counter</button>
      { showCounter && <Counter/> }
    </div>
  );
}
```

# Interval from effect still exists

Even after component is removed

- Adding component back creates extra effect
- This is why our count was upping by 2
    - Effect was run twice

We need to "clean up" our effect

# useEffect callback can return a function

This function is called when:

- component removed from page
- this useEffect called again

This function is used for "cleanup"

Example: if your effect created timeouts or intervals

- remove them because component and component state won't be there to update

# useEffect cleanup function

```
useEffect(
  () => {
    console.log('in effect', count);
    return () => {
      console.log('cleanup', count);
    };
  },
  [],
);
```

# Cleanup Counter

- To remove interval we need intervalId
  - But we don't want it in state
  - We use a **closure**
    - Reference to variable no longer in scope

```
useEffect(
  () => {
    const intervalId = setInterval( () => {
      console.log('incrementing');
      setCount(count => count + 1);
    }, 1000);
    return () => {
      console.log('cleanup');
      clearInterval(intervalId);
    };
  },
  [] // empty = effect on first render only
);
```

# Clean!

- We see the cleanup in the console
- Only counts by 1!
- Stops when component removed

Second render made problem more noticeable!

**Effects can cause problems when comp removed**

- Be sure to have cleanup for lasting effects
- Consider if component may no longer be there
    - For async effects
- Use the double-render in dev as a "canary"

# What is a Canary

From "Canary in a coal mine"

- Miners would take a caged bird with them
- Bird would show signs of bad air before humans
- Humans could leave before passing out/dying
    - Hopefully WITH the bird

Practices that help reveal problems early:

- **Canary**

# Summary - useEffect

A hook that takes a callback

- Callback runs after component renders
- Used for "side effects" to render
    - setup/cleanup needed for component

**Changing state in effect can cause infinite loop**

- Think about it before changing state

# Summary - Dependency Array

Second param to useEffect is a **dependency array**

- If not present
  - callback runs every render
- If present but empty (`[]`)
  - callback runs after first render only
- If present with values
  - callback runs if any values change
- If calling a state setter (avoid infinite loop!)
  - use function form to reference current value
  - avoid putting changing state in dep. array

# Summary - Cleanup function

The useEffect callback can return a function

- automatically used for **cleanup**
- remove timeouts/intervals
- disconnect any external effects

# Summary - Double Render in dev

React 18 does a double render in development

- Can reveal when effects aren't being cleaned up
- Only useful if you pay attention
    - Keep console clean
    - Deal with warnings and errors
    - Check console often