CoGrammar

Welcome to this session:

React – Hooks

The session will start shortly...

Questions? Drop them in the chat.
We'll have dedicated moderators
answering questions.

Safeguarding & Welfare

We are committed to all our students and staff feeling safe and happy; we want to make sure there is always someone you can turn to if you are worried about anything.

If you are feeling upset or unsafe, are worried about a friend, student or family member, or you feel like something isn't right, speak to our safeguarding team:

Ian Wyles
Designated Safeguarding Lead

Simone Botes

Nurhaan Snyman

Rafiq Manan

Ronald Munodawafa

Tevin Pitts

Scan to report a safeguarding concern

or email the Designated Safeguarding Lead:
Ian Wyles
safeguarding@hyperiondev.com

CoGrammar HyperionDev

Skills Bootcamp Cloud Web Development

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly. **(Fundamental British Values: Mutual Respect and Tolerance)**
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Academic Sessions. You can submit these questions here: **Questions**

Skills Bootcamp Cloud Web Development

- For all **non-academic questions**, please submit a query:
  www.hyperiondev.com/support
- **Report a safeguarding incident: *www.hyperiondev.com/safeguardreporting***
- We would love your feedback on lectures: Feedback on Lectures
- If you are hearing impaired, please kindly use your computer's function through Google chrome to enable captions.

Learning Outcomes

By the end of this lesson, learners should be able to:

- **Use useState and useEffect hooks** to manage data and side effects in React applications.
- **Implement useRef** for managing focus and DOM interactions.
- **Fetch and display data from external APIs** dynamically.
- **Design user-friendly interfaces** to present fetched data intuitively.

What is the main reason React introduced hooks?

A. To allow for better code reuse and composition.
B. To simplify state management in functional components.
C. To improve performance and optimize rendering.
D. To make React easier for beginners to learn.

What type of data can be stored in state with the useState hook in React?

A. Only primitive data types (e.g., strings, numbers, booleans).
B. Only arrays and objects.
C. Any type of data, including primitive values, arrays, and objects.
D. Only strings and numbers.

CoGrammar

Lecture Overview

- → Introduction to React hooks
- → State Hooks
- → Effect Hooks
- → Ref Hooks

CoGrammar

Introduction to React hooks

- ❖ React used to rely on class components, which could be cumbersome for building complex UIs.
- ❖ They were a hassle because you always had to switch between classes, higher-order components, and render props.
- ❖ Now all of this can be done without switching by using function components with React hooks.
- ❖ This means cleaner, easier-to-read code and a more enjoyable development experience.

CoGrammar

What are React hooks?

- ❖ Hooks are JavaScript functions that manage the state's behaviour and side effects by isolating them from a component.
- ❖ Before Hooks, class components were used, which allowed internal state to be managed and lifecycle events to be handled directly.
- ❖ React Hooks allow us to work with React components in a simpler and more concise way, without having to write classes.
- ❖ Hooks also make our code more readable and maintainable.

CoGrammar

React hook types

- ❖ There are several types of hooks used in React:
  - ➢ State hooks
  - ➢ Context hooks
  - ➢ Ref hooks
  - ➢ Effect hooks
  - ➢ Performance hooks
  - ➢ Additional hooks
  - ➢ Custom hooks

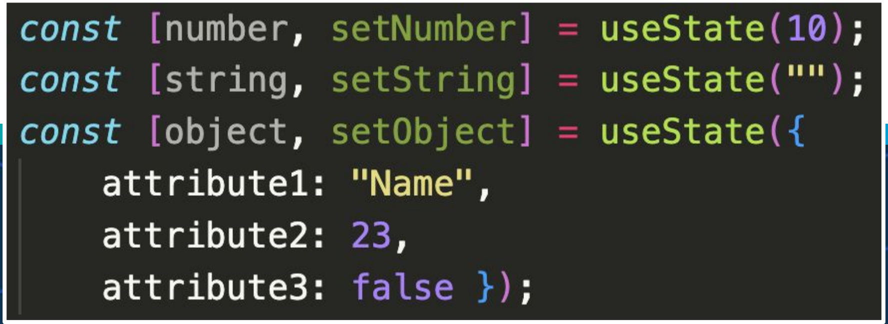
State Hook

❖ This hook is used for **state management**, allowing components to store and retrieve information.

➢ The **useState** hook declares a **state variable**, which is **preserved between function calls** and whose **change triggers a re-render**.

➢ The function **accepts** the **initial state** of the variable as input.

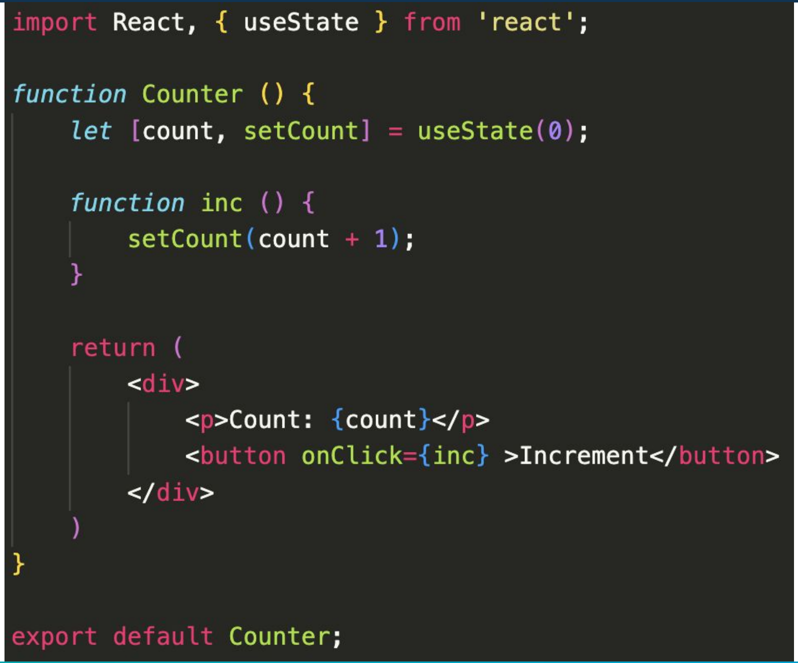
➢ The function **returns** a pair of values: the **state variable** and the **function** that updates it.



```
const [number, setNumber] = useState(10);
const [string, setString] = useState("");
const [object, setObject] = useState({
    attribute1: "Name",
    attribute2: 23,
    attribute3: false });
```

CoGrammar

Recap on Functional Components

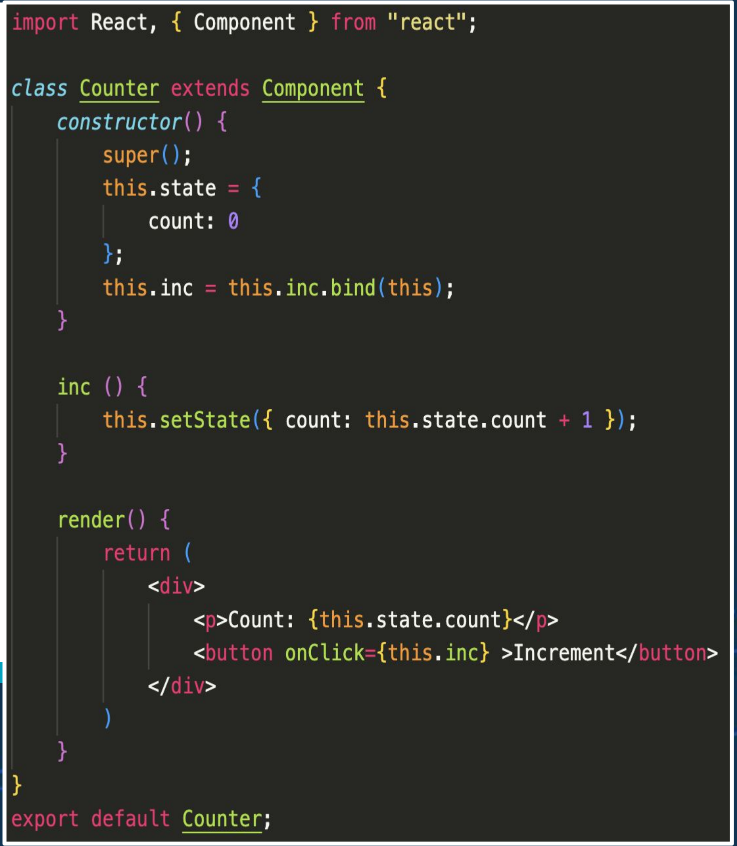


```
import React, { useState } from 'react';

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

    function inc () {
        setCount(count + 1);
    }

    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={inc} >Increment</button>
        </div>
    )
}

export default Counter;
```

CoGrammar

Example of a Class Component

- This is how we would implement the counter with a class component.



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

class Counter extends Component {
    constructor() {
        super();
        this.state = {
            count: 0
        };
        this.inc = this.inc.bind(this);
    }

    inc () {
        this.setState({ count: this.state.count + 1 });
    }

    render() {
        return (
            <div>
                <p>Count: {this.state.count}</p>
                <button onClick={this.inc} >Increment</button>
            </div>
        )
    }
}
export default Counter;
```

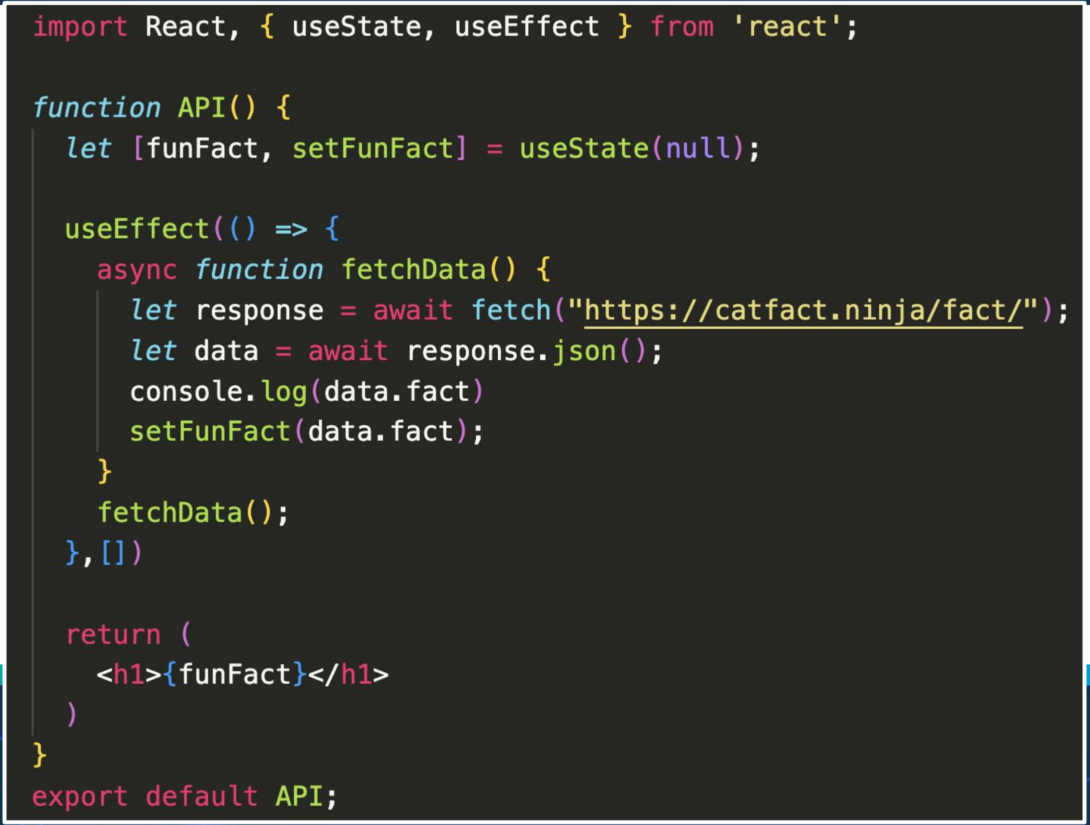
CoGrammar

Effect Hook

- This hook used for **connecting to and synchronizing external systems** after your components are rendered, known as performing side effects.
  - The **useEffect** hook is used for tasks like **fetching data**, **directly updating the DOM** and setting up **event listeners**.
  - The function takes in two arguments: a **block of code** which will be executed when the component is loaded, and a **dependencies list**, which is a list of variables whose change will trigger the first argument to be rerun.
  - If no **dependency argument** is passed, the first argument will run on **every render**.
  - If an **empty dependency argument** is passed, the first argument will on be run on the first render of the component.

CoGrammar

Fetch Data from API



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

function API() {
  let [funFact, setFunFact] = useState(null);

  useEffect(() => {
    async function fetchData() {
      let response = await fetch("https://catfact.ninja/fact/");
      let data = await response.json();
      console.log(data.fact)
      setFunFact(data.fact);
    }
    fetchData();
  },[])

  return (
    <h1>{funFact}</h1>
  )
}
export default API;
```

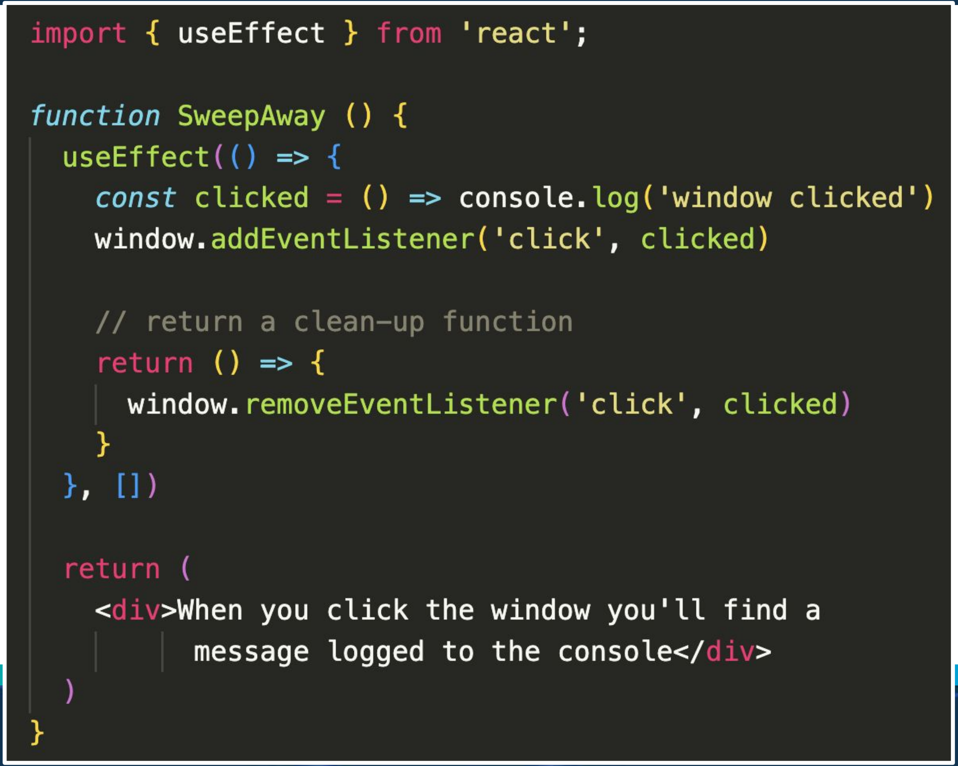
CoGrammar

Cleanup Function

- A function returned by the **useEffect** hook which gets executed before **every rerun** of the component and after the component is removed.
  - Tasks that can be performed in the **useEffect** hook, may need to be **aborted or stopped** when the **component is removed** or when **state changes**.
  - For example, API calls may need to be aborted, timers stopped and connections removed.
  - If this is not handled properly, your code may attempt to update a state variable which no longer exists, resulting in a **memory leak**.
  - This is done with a **cleanup function**, which is **returned by the useEffect hook**. This function will run when the component is **removed** or **rerendered**.

CoGrammar

Example of a Cleanup Function



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

function SweepAway () {
  useEffect(() => {
    const clicked = () => console.log('window clicked')
    window.addEventListener('click', clicked)

    // return a clean-up function
    return () => {
      window.removeEventListener('click', clicked)
    }
  }, [])

  return (
    <div>When you click the window you'll find a
        message logged to the console</div>
  )
}
```

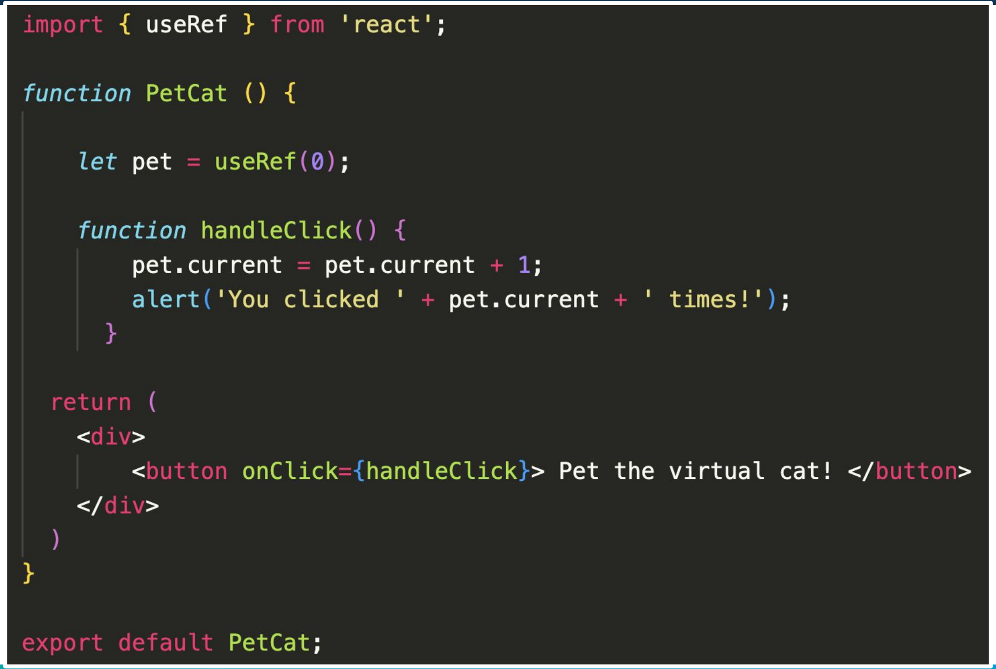
CoGrammar

Ref Hook

- ❖ This hook is used to store **mutable** values which **do not trigge**r re-renders and update DOM elements directly.
    - ➢ The **useRef** hook is store values which **persist between re-renders**, but d**o not cause the component to re-render** when changed.
    - ➢ We can also access DOM elements using useRef by passing the returned object to elements in the **ref** attribute.
    - ➢ The function accepts an **initial value** as an **input**.
    - ➢ The function returns an **object** with the property current initialised to the value passed as input to the function.

CoGrammar

Ref Hook



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

function PetCat () {

    let pet = useRef(0);

    function handleClick() {
        pet.current = pet.current + 1;
        alert('You clicked ' + pet.current + ' times!');
      }

  return (
    <div>
        <button onClick={handleClick}> Pet the virtual cat! </button>
    </div>
  )
}

export default PetCat;
```

CoGrammar

Ref and State comparison

refs	state
`useRef(initialValue)` returns `{ current: initialValue }` function that accepts props as an argument and returns a React element (JSX).	`useState(initialValue)` returns the current value of a state variable and a state setter function `([value, setValue])`.
Doesn't trigger re-render when you change it.	Triggers re-render when you change it.
Mutable: You can modify and update the current value outside of the rendering process.	Immutable: You must use the state setting function to modify state variables to queue a re-render.
You shouldn't read (or write) the current value during rendering.	You can read the state at any time. However, each render has its own snapshot of the state that does not change.

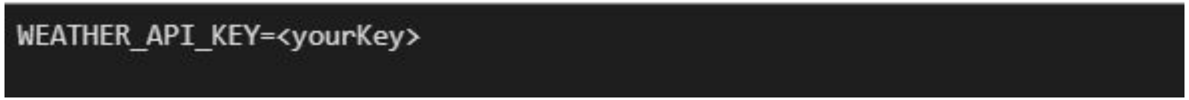
CoGrammar

.env file

- ❖ Storing sensitive information/data within your application is not really secure; for example, API keys.
- ❖ If you want to hide the key from your public code, it’s good practice to create a .env file to store these details.
- ❖ A .env file is like a secret storage locker for your project's important information that you don't want to share with everyone, such as passwords, API keys, or configuration settings. Instead of putting these details directly in your code (where anyone could see them if they look), you place them in the .env file.

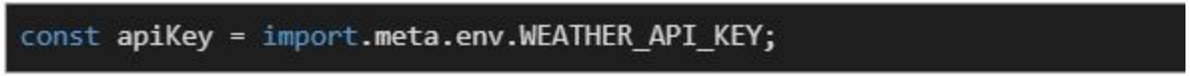
.env file example

- ❖ Add a file called .env in your root folder with key/pairs entries. For instance:



```
WEATHER_API_KEY=<yourKey>
```

- ❖ Now you can access the key stored in .env from anywhere in your React code by using the import.meta.env variable:



```
const apiKey = import.meta.env.WEATHER_API_KEY;
```

What is meant by a "side effect" in the useEffect React hook?

A. Any operation that updates the UI based on state changes.
B. Any operation that interacts with external systems, like APIs or timers.
C. Any operation that only updates local component variables.
D. Any operation that runs after the component renders.

What is the purpose of the useRef hook in React?

A. To manage state in functional components.
B. To perform side effects in a component.
C. To store a mutable reference to a DOM element or a value that persists across renders.
D. To trigger a re-render when the value changes.

CoGrammar

Let's take a break

CoGrammar

Questions and Answers

CoGrammar

Thank you for attending

CoGrammar

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education