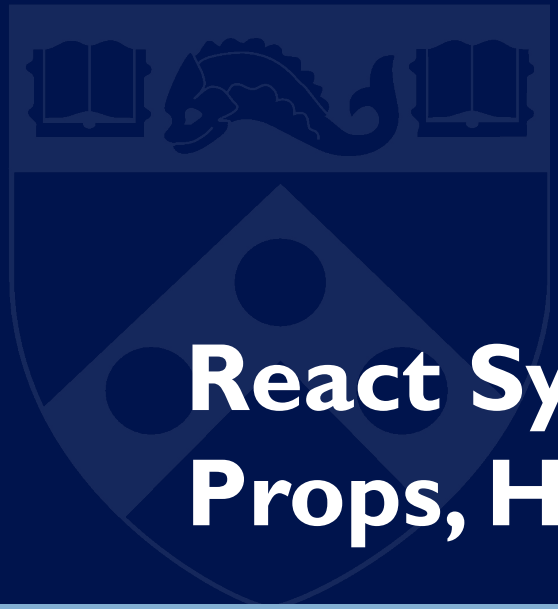
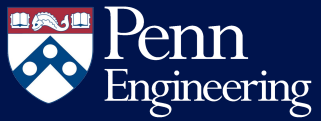


React Syntax: Components, Props, Hooks, Events, & Style

CIS 1962 (Fall 2025)
November 3rd, 2025



Logistics

- Homework 3 Due Today!
- Homework 4 Delayed!
- Final project released!
  https://upenn-cis-1962.vercel.app/final-project

Lesson Plan

5	JSX and Component Syntax
16	React Props
28	Conditional/List Rendering
37	Events & Inputs
43	React Hooks
64	Styling in React

PollEverywhere!

We will use PollEv to take attendance and do polls during lecture. Scan this QR Code or use the link to join for attendance! (Please add your name for identification)

PollEv.com/voravichs673

JSX and Component Syntax

What syntax do we use to create components in React?

Review: The "root" Container

Within a project's `index.html`, there is a container with the id "root" that React uses to render content to the page.

```
<div id="root"></div>
```

Your entry point to React will be some file that calls the `createRoot()` function from this HTML element, and a `render` function to define what will show up in that container:

```
import React from 'react';
import { createRoot } from 'react-dom/client';

createRoot(document.getElementById('root')).render(
     <p> Hello World! <p/>
)
```

JSX: JavaScript XML

JSX makes it easier for us to write React apps by converting HTML into React elements, allowing our scripts to create HTML.

```
const myElement =
document.createElement('p');
myElement.textContent = 'Hello World!';
document.getElementById('root').appendChild
(myElement);
```

Plain JS

```
const myElement = <p> Hello World! <p/>;

createRoot(document.getElementById('root'))
.render(
     myElement
)
```

React

JSX Syntax: Attributes & Styles

Common attributes like `class` are reserved by JS, so you have to use JSX versions like `className` instead.

```
<p class="highlight">I am made for plain JS, I use class!</p>
```

```
<p className="highlight">I am made for JSX, I use className!</p>
```

Additionally, any inline styles will use camelCase, rather than kebab-case, like their JS counterparts:

```
<p style={{ color: 'red', fontSize: '12px' }}>I'm some red text, size 12!</p>
```

JSX Syntax: Multiple Lines

A single JSX expression can represent multiple lines of HTML if wrapped in parentheses ().

However, there must be only **one** parent/top level element for each expression. Often, you'll use an fragment, a set of empty tags <></> to wrap items together.

```
const myElement = (
  <>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </>
);

createRoot(document.getElementById
('root')).render(
  myElement
);
```

JSX Syntax: Expressions

You can insert JavaScript expressions within JSX to execute JS code.

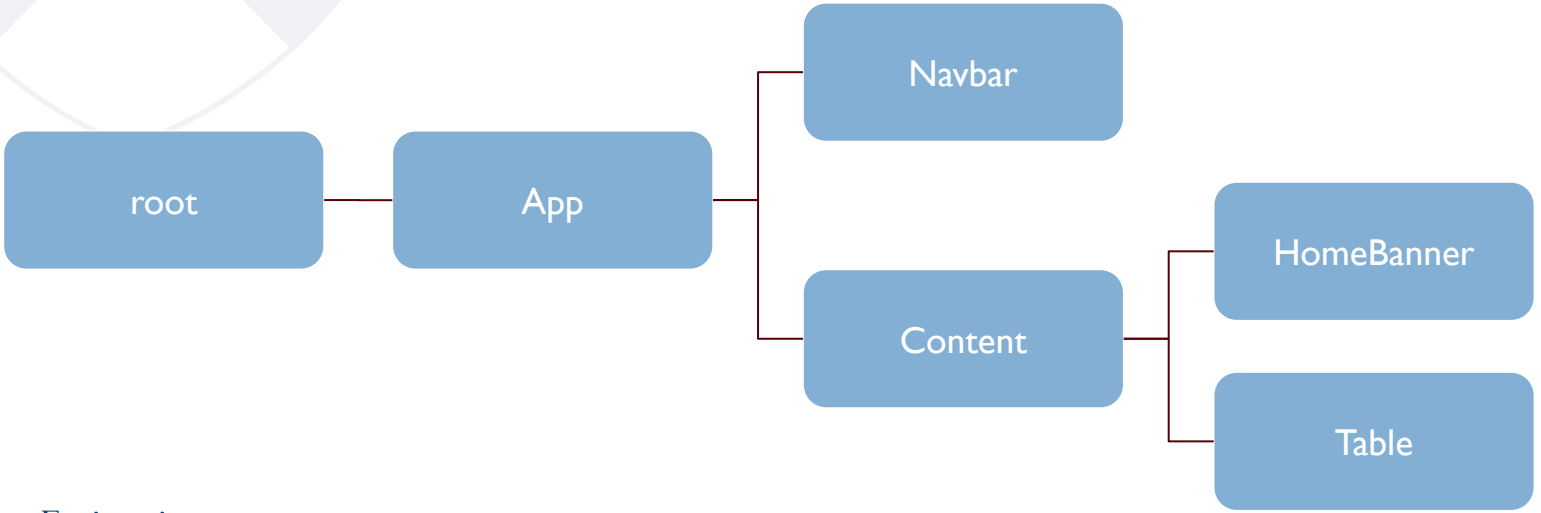
You define an expression by using the curly brackets {}

This allows you to use the breadth of JS syntax in the expression: variables, ternaries, string methods, and more!

```
<a className="button" href={window.loggedIn ? '/logout' : '/login'}>
    {window.loggedIn ? 'Log Out' : 'Log In'}
</a>
```

React Components

Components are the building blocks of React apps. We can represent a like a tree, that is made up of parent and child components.



Functional Components

Most React components use functional components as they are less complex and more top-to-bottom readable.

They often use hoisted **function declaration** (classic) or **arrow functions** (preferred in modern React).

```
function Hello(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

Hoisted functions

```
const Hello = (props) => {
  return <h1>Hello, {props.name}!</h1>;
};
```

Arrow functions (preferred)

Component Logic

Within functional components, before you return JSX, you will often perform rendering logic, and even embed other functions inside the component for later use.

```
const Fibonacci = ({ n }) => {
  function fib(num) {
    if (num <= 1) return num;
    return fib(num - 1) + fib(num - 2);
  }

  const result = fib(n);

  return <p>Fibonacci of {n} is {result}</p>;
}
```

Using Components

```
const Card = () => {
  return (
     <div className='card'>
        ...
     </div>
  );
};

export default Card;
```

```
import Card from './Card'

const CardGrid = () => {
  return (
     <div className='grid-squares'>
          <Card/>
          <Card/>
          <Card/>
          <Card/>
     </div>
  );
};
```

Once declared, components provide reusable ways to insert JSX elements into other JSX elements.

Often, you'll have components declared in their own file and imported into other components where they are needed.

POLL QUESTION

Debug: What's wrong with this component?

```jsx
import { useState } from "react";

const SimpleCard = ({ title, description }) => {
  const [liked, setLiked] = useState(false);

  return (
    <div className="bg-white p-4 rounded-lg shadow-md max-w-xs">
      <h2 className="text-lg font-bold mb-2">{title}</h2>
      <p className="text-gray-600 mb-4">{description}</p>
      <button
        className={`text-xl ${liked ? "text-red-500" : "text-gray-400"}`}
        onClick={() => setLiked((prev) => !prev)}
      >
        ♥
      </button>
    </div>
    <div className="text-xs text-gray-400 mt-2">Click the heart to like!</div>
  );
}
export default SimpleCard
```

Two parent elements (2 divs)!

React Props

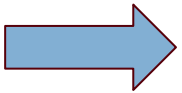
How do we pass data from parent to child components within React?

What are Props?

Props, short for "properties", are the arguments passed into React components. These take the form of the actual arguments of the functional components you write.

These props are read-only inside the component that receives them, representing a unidirectional data flow from parent to child.

```jsx
// App.jsx
<Hello name="Voravich" />
```



```jsx
const Hello = (props) => {
  return <h1>Hello, {props.name}!</h1>;
};
```

Passing and Using Props

In some parent component, you can pass props as **attributes** on JSX elements.

Then in the child component, you access the props via the `props` object parameter, or **destructuring** the `props` object.

```
import Button from "./Button"

const ButtonExample = () => {
  return (
     <>
          <Button label="Click me"
                  color="red" />
     </>
  );
};
```

```
const Button = ({ label, color }) => {
  return <button style={{ background:
color }}>{label}</button>;
}
```

Functions as Props

Since prop data is immutable in the child, we need alternative ways for the child to communicate with with the parent.

Callback functions as props provide ways to children to communicate changes with the parent component.

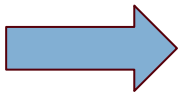
```
const Parent = () => {
  const handleClick = () => alert('Button clicked!');
  return <Child onClick={handleClick} />;
}

const Child = ({ onClick }) => {
  return <button onClick={onClick}>Click Me</button>;
}
```

Wrapper Components

You can access nested JSX content between the opening and closing tags of one component to another with `props.children` property.

This is useful for dynamic rendering and layout of certain components.

```
<Card>
  <h2>Title</h2>
  <p>Description</p>
</Card>
```



```
const Card = ({ children }) => {
  return <div
className="card">{children}</div>;
}
```

Prop Drilling

```
const Grandparent = () => {
  return <Parent user="voravich" />;
}

const Parent = (props) => {
  return <Child user={props.user} />;
}

const Child = (props) => {
  return <div>User: {props.user}</div>;
}
```

In some cases you may want to pass data across several nested layers of parent/child, leading to **prop drilling**.

This may become unwieldy, and could be solved using the hook useContext instead.

Props and TypeScript

With props, it's important to maintain type safety, since it is one of the ways data is passed through a React app.

With TypeScript you can used typed props in components:

```tsx
interface ButtonProps {
  label: string;
  color?: string;
  onClick: () => void;
}

const Button: React.FC<ButtonProps> = ({ label, color = "blue", onClick }) => (
  <button style={{ background: color }} onClick={onClick}>
    {label}
  </button>
);
```

TS Types for React

React.FC<P>: Functional component with typed props

React.ReactNode: Any valid renderable content (elements, strings, numbers, fragments, etc.)

React.ReactElement: The object returned from JSX

React.ChangeEvent<T>: Form/input event type

React.MouseEvent<T>: Mouse event type

Optional/Required and Default Props

By default, props are optional: if a prop is not provided in the parent, it will be undefined for its value in the child.

We can provide a default prop value in the case no values are provided.

TS can also allow us to make prop values required/optional.

```
interface MessageProps { text?: string; }

const Message: React.FC<MessageProps> = ({ text = "Hello!" }) => <div>{text}</div>;
```

Props.children and Generics

This pattern below can provide more flexibility when you are using `props.children` in your components with TypeScript:

```tsx
interface CardProps {
  children: React.ReactNode;
}

const Card: React.FC<CardProps> = ({ children }) =>
<div>{children}</div>;
```

POLL QUESTION

Debug: What's TypeScript error is present in this component?

```tsx
const Notice = ({ title, subtitle }: {title: string, subtitle?: string} ) => (
  <section>
    <h1>{title}</h1>
    <h2>{subtitle}</h2>
  </section>
);

const Wrapper = ({ message }: {message: string}) => (
  <div>
    <Notice subtitle="This is a subtitle" />
    <p>{message}</p>
  </div>
);

const App = () => <Wrapper message="All good!" />;

export default App
```

Property 'title' is missing in type '{ subtitle: string; }' but required in type '{ title: string; subtitle?: string | undefined; }'.

Conditional & List Rendering

How we we leverage JSX to easily render items under certain conditions?

Conditional Rendering

A common pattern in React is using conditional logic to only render items under certain conditions.

This often gets used with the useStates hook to have automatic updates to components.

Plain HTML/JS Conditionals

With plain HTML/JS, you would need to manually update the DOM or trigger updates with eventListeners.

Messy, prone to bugs, hard to scale!

```
const user = { isLoggedIn: true, name: "Alice" };
const appDiv = document.getElementById("app");

function render() {
  if (user.isLoggedIn) {
    appDiv.textContent = `Welcome, ${user.name}!`;
  } else {
    appDiv.textContent = "Please log in";
  }
}

render()
```

React Conditionals

With React, we can automatically re-render items on state changes.

Cleaner and more scalable!

```
const Welcome = ({ user }) => {
  return (
    <div>
      {user.isLoggedIn ? `Welcome,
${user.name}!` : "Please log in"}
    </div>
  );
}
```

```
const App = () => {
  const [user, setUser] = useState({ isLoggedIn:
true, name: "Alice" });

  const toggleLogin = () => {
    setUser((prevUser) => prevUser.isLoggedIn
      ? { isLoggedIn: false, name: "" }
      : { isLoggedIn: true, name: "Alice" }
    );
  };

  return (
    <div>
      <button onClick={toggleLogin}>
        {user.isLoggedIn ? "Log out" : "Log in"}
      </button>
      <Welcome user={user} />
    </div>
  );
}
```

Conditionals with if/ternary

You will often use if statements (outside of JSX) or ternary operators (within JSX) to do conditional rendering.

```
const Greeting = ({ isMember, name }) => {
  return (
    <div>
      {
        isMember
          ? <h2>Welcome back, {name}!</h2>
          : <h2>Hello, guest. Please sign up.</h2>
      }
    </div>
  );
}
```

Conditionals with &&

Leveraging truthy/falsy values, we can allow something to be rendered only when some value exists or is truthy.

```
const Notification = ({ message }) => {
  return (
    <div>
      {message && <div className="notification">🔔 {message}</div>}
    </div>
  );
}
```

Conditionals with Switch

Switch cases can allow you to handle more than 2 cases.

```
const StatusMessage = ({ status }) => {
  switch (status) {
    case "loading": return <Spinner />;
    case "error": return <Error />;
    case "success": return <Success />;
    default: return null;
  }
}
```

List Rendering

JSX makes it easy to dynamically render an entire list of items as multiple JSX elements.

Each list element needs a **unique key** so that React can identify elements for updates and optimization.

```jsx
function UserList() {
  const users = [
    { id: 1, name: "Alice" },
    { id: 2, name: "Bob" },
    { id: 3, name: "Charlie" },
    { id: 4, name: "Diana" }
  ];

  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

POLL QUESTION

Debug: Identify the 2 errors in the following code.

```tsx
function App() {
  const orders = [
    { id: 101, customer: "Alice", items: ["Book", "Pen"], total: 12.9 },
    { id: 102, customer: "Bob", items: ["Notebook"], total: 4.2 },
    { id: 103, customer: "Carol", items: ["Eraser", "Book"], total: 3.3 },
  ];

  return (
    <section>
      <h2 className="font-bold text-lg">Order History</h2>
      <ul>
        {orders.map(order =>
          <li className="mb-3" key={order.id}>
            <div>Customer: {order.customer}</div>
            <ul>
              {order.items.map(item =>
                <li key={order.id}>
                  <span>{item}</span>
                </li>
              )}
            </ul>
          </li>
          <p>Total: ${order.total}</p>
        )}
      </ul>
    </section>
  );
}
```

1. Two parent elements in orders.map
2. Key in orders.map and order.items.map is the same

POLL QUESTION: Corrected

Debug: Identify the 2 errors in the following code.

```tsx
function App() {
  const orders = [
    { id: 101, customer: "Alice", items: ["Book", "Pen"], total: 12.9 },
    { id: 102, customer: "Bob", items: ["Notebook"], total: 4.2 },
    { id: 103, customer: "Carol", items: ["Eraser", "Pencil"], total: 3.3 },
  ];

  return (
    <section>
      <h2 className="font-bold text-lg">Order History</h2>
      <ul>
        {orders.map(order =>
          <li className="mb-3" key={order.id}>
            <div>Customer: {order.customer}</div>
            <ul>
              {order.items.map(item =>
                <li key={`${order.id}-${item}`}>
                  <span>{item}</span>
                </li>
              )}
            </ul>
            <p>Total: ${order.total}</p>
          </li>
        )}
      </ul>
    </section>
  );
}
```

1. Two parent elements in orders.map
2. Key in orders.map and order.items.map is the same

Events and Inputs

How do we handle user actions and inputs in React?

React Event Handling

React can handle events similar to HTML DOM events.

There's a few notable differences:

- Events are written in camelCase (onClick, not onclick)
- Handlers are often written as functions called inside of JSX.

```
const MyButton = () => {
  function handleClick() {
    alert("You clicked me!");
  }

  return <button onClick={handleClick}>Click Me</button>;
}
```

Common React Event Listeners

onClick: User clicks an element (button, div, etc.)

onChange: User input changes (input, textarea, select)

onSubmit: Form submission

onMouseEnter / onMouseLeave: Mouse moves in/out of an element

onKeyDown / onKeyUp: User presses keyboard keys

onFocus / onBlur: Input focus handling

Forms and Inputs

React forms are **controlled components**, meaning input values are synced with the component's state, not the DOM.

Forms often use `onSubmit`, while inputs often use `onChange` to manage updates.

Don't forget `preventDefault`!

```
const MyForm = () => {
  const [name, setName] = useState("");

  function handleChange(e) {
    setName(e.target.value);
  }

  function handleSubmit(e) {
    e.preventDefault();
    alert(`Hello, ${name}!`);
  }

  return (
    <form onSubmit={handleSubmit}>
      <input value={name} onChange={handleChange}
placeholder="Your name" />
      <button type="submit">Say hello</button>
    </form>
  );
}
```

POLL QUESTION

What happens when the user clicks directly on the "Increment" button?

```tsx
import { useState } from "react";

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

  function increment(e) {
    setCount(count + 1);
    e.stopPropagation();
  }

  return (
    <div
      onClick={() => setCount(count - 1)}
      style={{ border: "1px solid black", padding: 20 }}
    >
      <button onClick={increment}>Increment</button>
      <p>{count}</p>
    </div>
  );
}
```

The counter increases by 1.

5-Minute Break!

Hooks

What are React Hooks and what powerful features do they provide?

Early React: Class Components

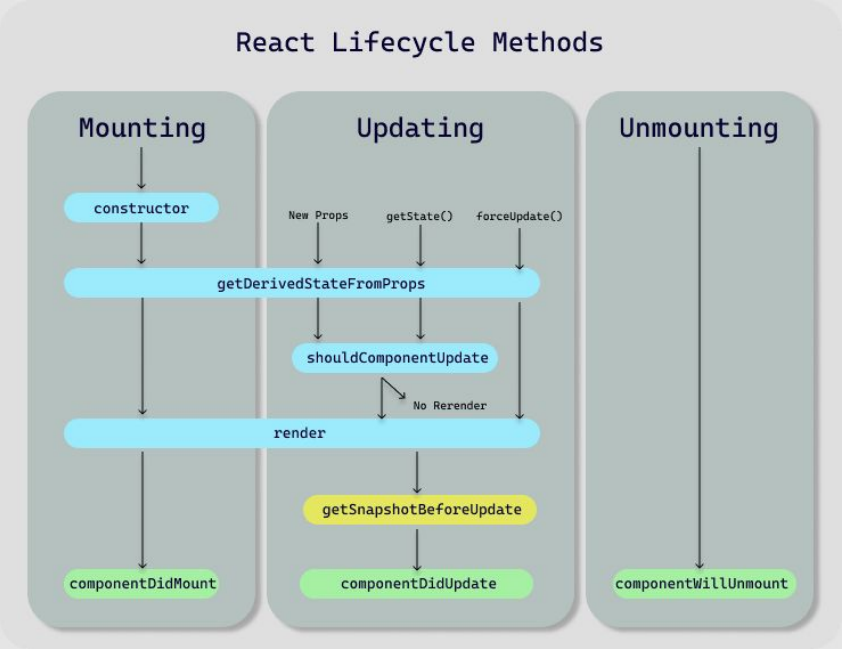
Before Hooks, **class components** were used over functional components due to them allowing state management.

However, it was quite complex, required specific syntax (`this` binding), and some component lifecycle management.

```jsx
import React from "react";

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

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

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.handleIncrement}>Increment</button>
      </div>
    );
  }
}

export default Counter;
```

Component Lifecycle

There used to have some annoying lifecycle methods to tell React **when** to do things, like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.



React Hooks

In 2019, **Hooks** were introduced to React, allowing functional components to have states.

This made managing states simpler with functional component syntax without needing to manage lifecycle methods!

After Hooks, functional components became standard while class components were phased out.

Component Lifecycle, Simplified

While the component lifecycle still exists, Hooks make it easier to interface with it.

- The body of functional components runs every re-render, but any stateful values are kept between renders
- useEffect covers most of the lifecycle methods in terms of mounting, updating, and cleanup with the use of a dependency array.

Using Hooks

Hooks need to be imported from the React library to be used.

They are named exports within the library.

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

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

  return (
    <button onClick={() => setCount(count + 1)}>
      Count: {count}
    </button>
  );
}
```

useState: Initialize

`useState` accepts an initial state and returns 2 values:

- The current state
- An update function

You'll often use destructuring to get these 2 values.

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

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

  return (
    <button onClick={() => setCount(count + 1)}>
      Count: {count}
    </button>
  );
}
```

useState: Read and Update

You can use the state variable anywhere in the scope of the component.

To update, you can use the state updater function only, you cannot **directly** update the state variable value.

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

const Counter = () => {

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

  return (
    <button onClick={() => setCount(count + 1)}>
      Count: {count}
    </button>
  );
}
```

useState: Updating Objects

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

const UserProfileForm = () => {
  const [user, setUser] = useState({
    name: "Jane Doe",
    preferences: {
      newsletter: true
    }
  });

  const handleNewsletter = () => {
    setUser((prevUser) => ({
      ...prevUser,
      preferences: {
        ...prevUser.preferences,
        newsletter:
!prevUser.preferences.newsletter
      }
    }));
  };

...
```

```
 ...

  return (
    <div>
      <h2>User Profile</h2>
      <p>Name: {user.name}</p>
      <label>
        Subscribe to newsletter:
        <input
          type="checkbox"
          checked={user.preferences.newsletter}
          onChange={handleNewsletter}
        />
      </label>
      <pre>{JSON.stringify(user, null, 2)}</pre>
    </div>
  );
}

export default UserProfileForm;
```

Updates are Batched and Async

For performance sake, multiple updates to states in a short period are **batched** together to re-render at the same time.

State updates are also **asynchronous**, being scheduled for the most optimization rather than occurring synchronously the moment their update functions are called.

POLL QUESTION

What sequence of values outputs to the alert dialogs if a user presses the button twice rapidly?

```tsx
import  { useState } from "react";

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

  function handleAlertAdd() {
    setTimeout(() => {
      setCount(count + 1);
      alert(count);
    }, 1000);
  }

  return (
    <div>
      <p>{count}</p>
      <button onClick={handleAlertAdd}>Add With Alert</button>
    </div>
  );
}
```

`0, then 0`

useEffect: Side Effects

Side effects are tasks unrelated to rendering UI, such as data fetching, timers, and logging.

`useEffect` is often used to unify lifecycle methods to run tasks on component mount, update, and unmount.

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

useEffect(() => {
    ...
});
```

useEffect: Dependency Array

The timing of when side effects run is controlled by the 2nd argument to `useEffect`: the dependency array.

Side effects can:

```
useEffect(() => {
    ...
});
```

Run every render

```
useEffect(() => {
    ...
}, []);
```

Run on first render

```
useEffect(() => {
    ...
}, [x, y]);
```

Run on first render AND When dependency (x or y) changes

useEffect: Timer

This component counts up continuously, since count is always changing and re-rendering.

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

  useEffect(() => {
    setTimeout(() => {
      setCount((count) => count + 1);
    }, 1000);
  });

  return <h1> {count} </h1>;
}
```

useEffect: CountOnce

This component counts up only once, on the first render.

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

  useEffect(() => {
    setTimeout(() => {
      setCount((count) => count + 1);
    }, 1000);
  }, [])

  return <h1> {count} </h1>;
}
```

useEffect: Counter

This component initially sets the calculation variable, then automatically runs again when count is updated.

```
const Counter = () => {
  const [count, setCount] = useState(0);
  const [calculation, setCalculation] = useState(0);

  useEffect(() => {
    setCalculation(() => count * 2);
  }, [count]);

  return (
    <>
      <p>Count: {count}</p>
      <button onClick={() => setCount((c) => c + 1)}>+</button>
      <p>Calculation: {calculation}</p>
    </>
  );
}
```

useEffect: Cleanup

If you want to perform actions when a component is unmounted or if the effect is rerun, you can return a function to perform the task.

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

  useEffect(() => {
    let timer = setTimeout(() => {
      setCount((count) => count + 1);
    }, 1000);

    return () => clearTimeout(timer);
  });

  return <h1> {count} </h1>;
}
```

POLL QUESTION

Debug: There is a bug in this code. What happens when you use this component?

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

function App({ search }) {
  const [results, setResults] = useState([]);

  useEffect(() => {
    fetch(`/api/suggest?q=${search}`)
      .then(res => res.json())
      .then(data => setResults(data));
  }, [results]);

  return (
    <ul>
      {results.map(item => <li key={item}>{item}</li>)}
    </ul>
  );
}
```

This component will run into an infinite loop of fetching and re-rendering.

useContext

`useContext` allows a way to manage state globally.

This solves the problem of prop drilling by managing a Context instead.

```
const Grandparent = () => {
  return <Parent user="voravich" />;
}

const Parent = (props) => {
  return <Child user={props.user} />;
}

const Child = (props) => {
  return <div>User: {props.user}</div>;
}
```

```
const UserContext = createContext();

const Grandparent = () => {
  return (
    <UserContext.Provider
value="voravich">
      <Parent />
    </UserContext.Provider>
  );
};

const Parent = () => {
  return <Child />;
};

const Child = () => {
  const user = useContext(UserContext);
  return <div>User: {user}</div>;
};
```

useRef

`useRef` allows values to persist between renders.

Any change to this value will not trigger a re-render.

```
const Toggle = () => {
  const [toggle, setToggle] = useState(true);
  const count = useRef(0);

  useEffect(() => {
    count.current = count.current + 1;
  });

  return (
    <>
      <button onClick={() =>
setToggle(!toggle)}>Press Me!</button>
      <p>Renders: {count.current}</p>
    </>
  );
}
};
```

useMemo

useMemo can be used to memorize values, essentially caching them.

This can be used to keep expensive operations from running every render, and only updating when the dependency updates.

```
function slowFunction(num) {
  console.log("Calculating...");
  let result = 0;
  for (let i = 0; i < 1e8; i++) {
    result += num;
  }
  return result;
}

const ExpensiveCalculator = ({ number }) => {
  const result = useMemo(() =>
slowFunction(number), [number]);

  return <div>Result: {result}</div>;
}
```

Styling in React

How do we write style code in React?

React Styles

We can write CSS the classic way with:

- **CSS Modules**
- **CSS-in-JS** with styled-components
- **Inline Styles and Utility Classes**

We can also make use of popular CSS and React UI Libraries:

- **CSS Libraries** like Bootstrap & Bulma
- **React UI** like MUI, React Bootstrap, & Chakra

CSS Modules

If you still want to write plain CSS, CSS Modules will allow you to classes scoped to components. This reduces collisions, and allows you to manage one CSS file per component easily.

```
import styles from './style.module.css';
// or import { myClass } from
"./style.module.css";

// function MyComponent(...) {
return <div
className={styles.myClass}>Hi!</div>;
```

```
/* style.css */

.myClass {
  font-size: 20rem;
}
```

CSS-In-JS

Alternatively, you can use a technique called CSS-In-JS where you write CSS directly in your JS code.

This keeps all your code in one place rather than in multiple files.

Additionally, since we're working with components, we extract repeated items into components for better readability!

CSS-In-JS: styled-components

`styled-components` is a popular library that allows you to insert styles right into JS.

We can define these style components with specific CSS styles and can insert them into the file immediately.

```
const Wrapper = styled.section`
  padding: 4em;
  background: papayawhip;
`;

const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: ${(props) => props.color ??
'#bf4f74'};
`;

// function MyComponent(...) {
return (
  <Wrapper>
    <Title color="#ffccff">How Stylish</Title>
  </Wrapper>
);
```

Inline styles with Utility Classes

Writing styles inline is also a popular method, through libraries like Tailwind.

It's faster than writing full CSS properties, and many libraries will include utility classes to easily build UIs, including pseudo-classes and mobile-responsive support.

```
<div
     className={`h-full flex ${active ?"justify-center items-end lg:items-center
      lg:justify-end" : ""}`}
>
     ...
</div>
```

CSS Libraries

In contrast to writing traditional CSS or Tailwind utility classes, CSS libraries will include a lot more opinionated styles and components for different themes with little customization.

```
// Bulma
const BulmaNotification = ({ message,
type = "is-primary" }) => (
  <div className={`notification
${type}`}>
    {message}
  </div>
);
```

```
<BulmaNotification type="is-danger"
message="Something went wrong!" />
```

React UI Libraries

You can go further by installing a React UI library that provides ready-made components.

You provide props to these components that under the hood that turned into style classes.

```
// MUI
<Grid xs={12} sm={12} md={6} lg={3} xl={3}>
  <Box
    width="100%"
    backgroundColor={colors.primary[400]}
    display="flex"
    alignItems="center"
    justifyContent="center"
  >
    <StatBox
      title="32,441"
      subtitle="New Clients"
      progress="0.30"
      increase="+5%"
      icon={
        <PersonAddIcon
          sx={{ color: colors.greenAccent[600], fontSize: '26px' }}
        />
      }
    />
  </Box>
</Grid>
```

Popular UI Libraries

Utility-First CSS Libraries

- Tailwind ❤️
- Emotion
- UnoCSS
- WindiCSS

CSS-In-JS

- styled-components
- vanilla-extract
- Stiches

CSS Libraries

- Bootstrap
- Bulma
- Foundation
- Semantic UI
- Materialize
- Skeleton

React UI Libraries

- Material UI (MUI)
- React Bootstrap
- Chakra UI
- Ant Design
- … and many more!