A general overview of what React.js is:

- Powerful JavaScript library for building user interfaces
- Created by FaceBook
- Uses JavaScript

React apps are made up of small, reusable pieces of code called **components**. Each component is responsible for rendering a part of the user interface, which makes it easy to build and maintain large applications.

React uses a lightweight copy of the DOM (Document Object Model) called the **Virtual DOM** to improve performance.
When changes occur, React updates only the necessary parts of the real DOM by comparing the virtual representation with the current state.

React is declarative which is separate from imperative which is what we had just gone over in class.

Declarative programming focuses on **what** you want to achieve without specifying **how** to do it.

- You describe the desired result, and the underlying system handles the implementation details.
- The emphasis is on the **end state** or **logic** of the computation

React's design maps closely to functional programming concepts that we've studied in this course:

1. **Functions as Values:** Components are just functions that return JSX. This mirrors the idea of treating functions as reusable, composable values.

2. **Higher-Order Functions:** React uses **higher-order functions** frequently. A higher-order function is a function that takes another function as input or returns a function as output. Higher-order functions, like `map` and `filter`, are common in both functional programming and React for building and manipulating data structures

3. **Immutability:** React encapsulates computation and handles state immutably, which is a key functional concept. Functional programming emphasizes **immutability**—React enforces this principle through props and state. Props: Components cannot modify their props, ensuring unidirectional data flow. State: State updates in React do not mutate the original state but create a new one.

By exploring React, we can see how these theoretical ideas come to life in modern software development.

At the top of the file, we import `React` and the `useState` hook from the React library. These are essential for creating functional components and managing state.

Next, we define our `Counter` component as a function. In React, functional components are just JavaScript functions that return JSX, which describes what the UI should look like.

Here, we declare the `count` state using the `useState` hook. This gives us two things:

- **`count`, which holds the current value of the counter.**
- **`setCount`, a function to update the `count` value.**

Next, we return the JSX that defines the structure and appearance of our app. This includes:

- **A title,**
- **The current count displayed dynamically,**
- **Three buttons for incrementing, decrementing, and resetting the count.**

The value of `count` is displayed dynamically in the `<h2>` tag. Every time we update `count` using `setCount`, React automatically re-renders the component to reflect the new value.

Each button has an `onClick` event handler that specifies what happens when the button is clicked. For example:

- **The Increment button increases the count by 1.**

- **The Decrement button decreases it by 1.**
- **The Reset button sets it back to 0.**

Finally, we export the `Counter` component so it can be used in other parts of our app