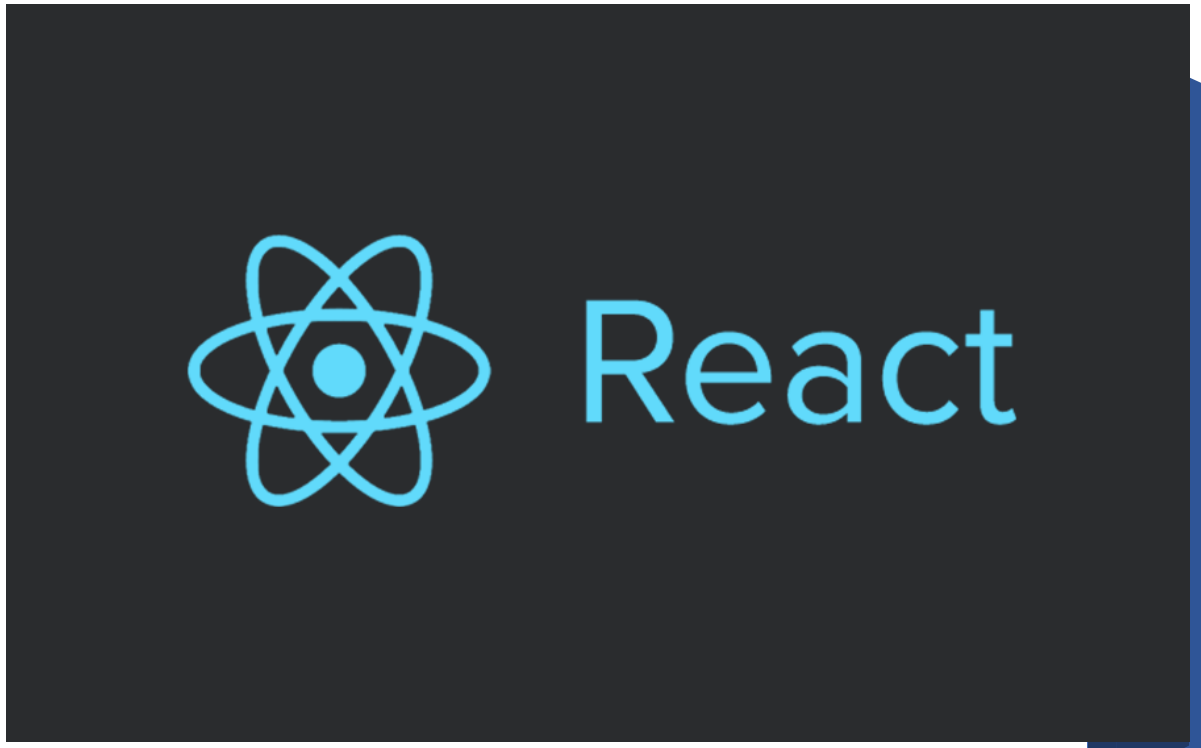
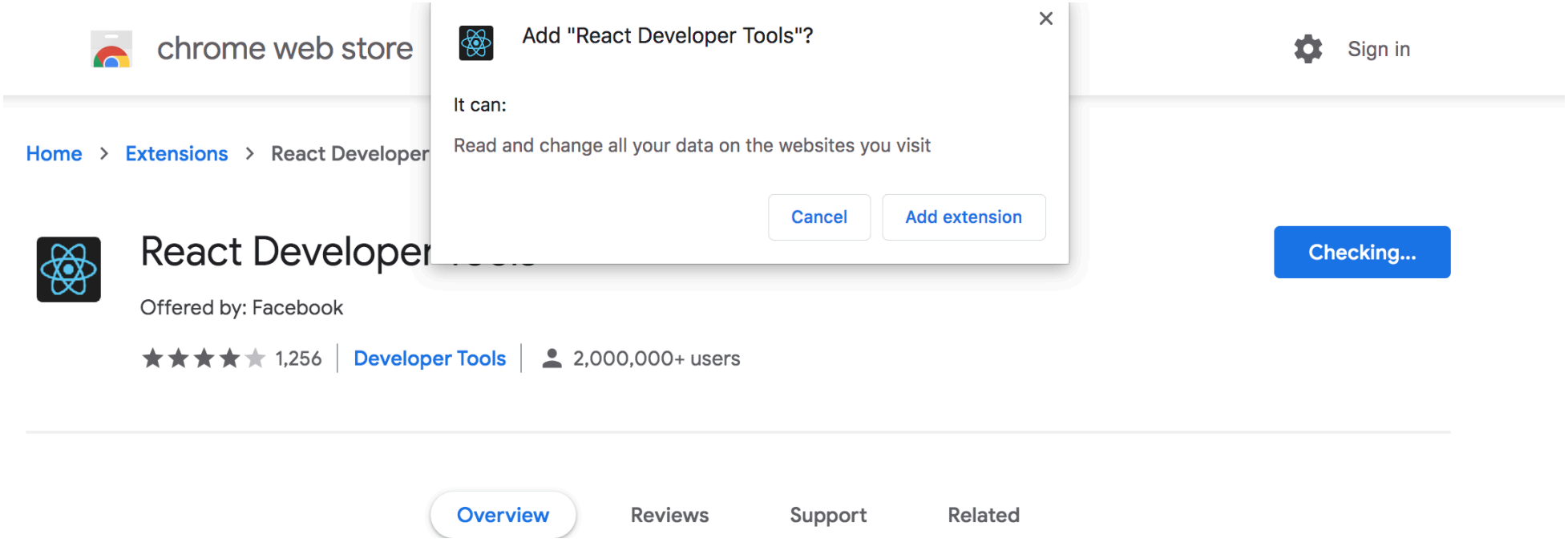


Full Stack Development - I

COMP 3124

ReactJS Developer Tools

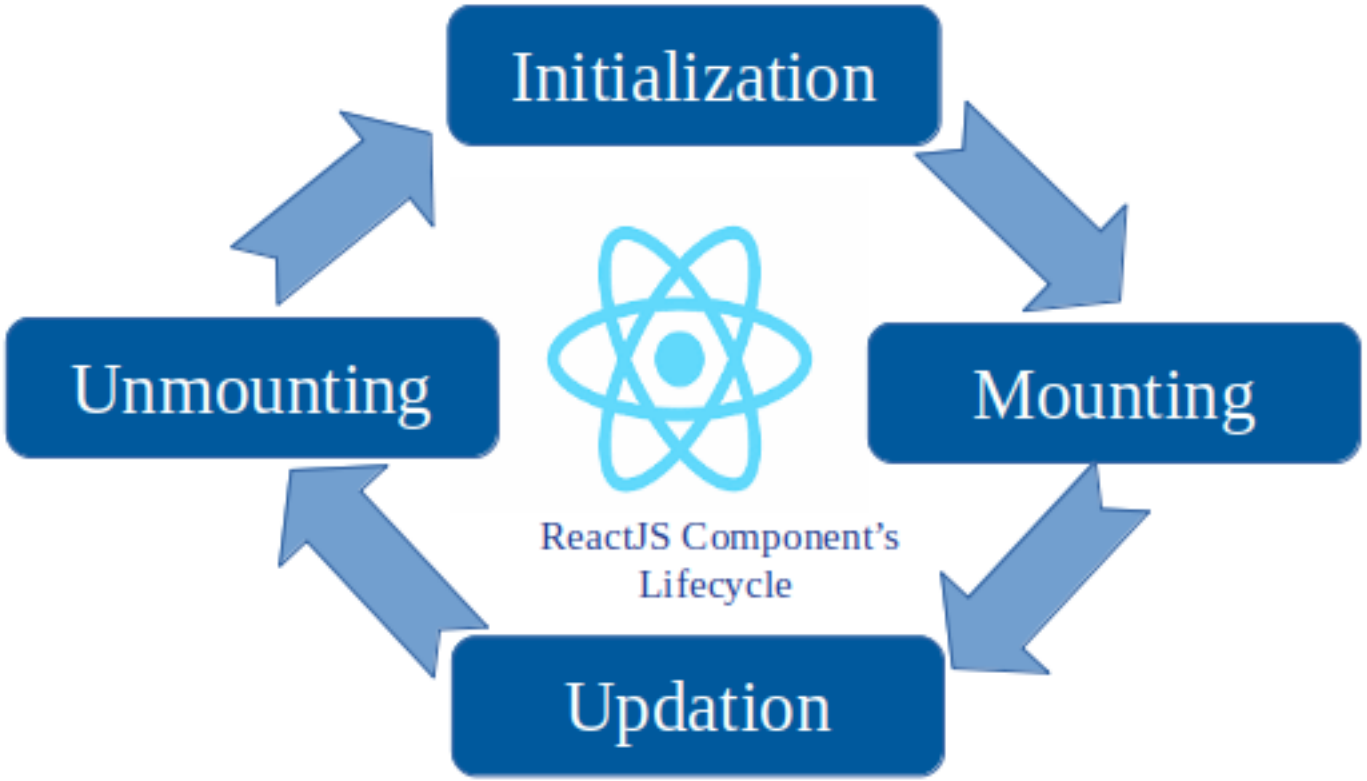
- The React Developer Tools is a plugin for the Chrome and Firefox browser.
- When you add the extension, you are adding additional tools to the developer console.
- Visit the Chrome plugin page for React Developer Tools to install the extension.
- Now that you are on a website that uses React, open the console to access the React Developer Tools.
- Open the console by either right-clicking and inspecting an element or by opening the toolbar by clicking **View > Developer > JavaScript console**.
- When you open the console, you'll find two new tabs: **Components** and **Profiler.**



ReactJS - Component Life Cycle

Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.

The three phases are: **Mounting**, **Updating**, and **Unmounting**.



ReactJS - Component Life Cycle

Lifecycle Hooks Methods

- **componentWillMount** is executed
  - before rendering, on both the server and the client side.
- **componentDidMount** is executed
  - after the first render only on the client side.
  - This is where AJAX/Network requests and DOM or state updates should occur.
  - This method is also used for integration with other JavaScript frameworks and any functions with delayed execution such as **setTimeout** or **setInterval**.
  - We are using it to update the state so we can trigger the other lifecycle methods.
- **componentWillReceiveProps** is invoked
  - as soon as the props are updated before another render is called.
- **shouldComponentUpdate** should return **true** or **false** value.
  - This will determine if the component will be updated or not.
  - This is set to **true** by default. If you are sure that the component doesn't need to render after **state** or **props** are updated, you can return **false** value.
- **componentWillUpdate** is called just before rendering.
- **componentDidUpdate** is called just after rendering.
- **componentWillUnmount** is called after the component is unmounted from the dom.

ReactJS- Component Events

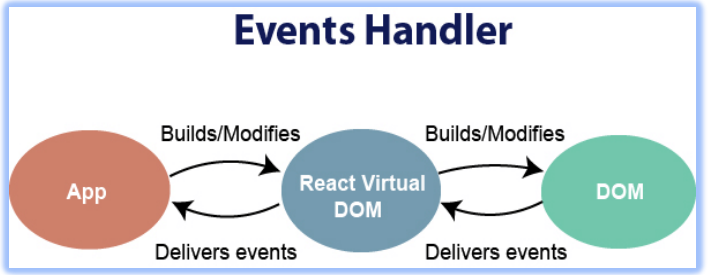
- An event is an action that could be triggered as a result of the user action or system generated event.
- For example, a mouse click, loading of a web page, pressing a key, window resizes, and other interactions are called events.
- Handling events with react have some syntactic differences from handling events on DOM.
    1. React events are named as **camelCase** instead of **lowercase**.
    2. With JSX, a function is passed as the **event handler** instead of a **string.**

Event declaration in plain HTML:

```
<button onclick="showMessage()">
    Hello JavaTpoint
</button>
```

Event declaration in React:

```
<button onClick={showMessage}>
    Hello JavaTpoint
</button>
```



React events are written in camelCase syntax:

`onClick` instead of `onclick`.

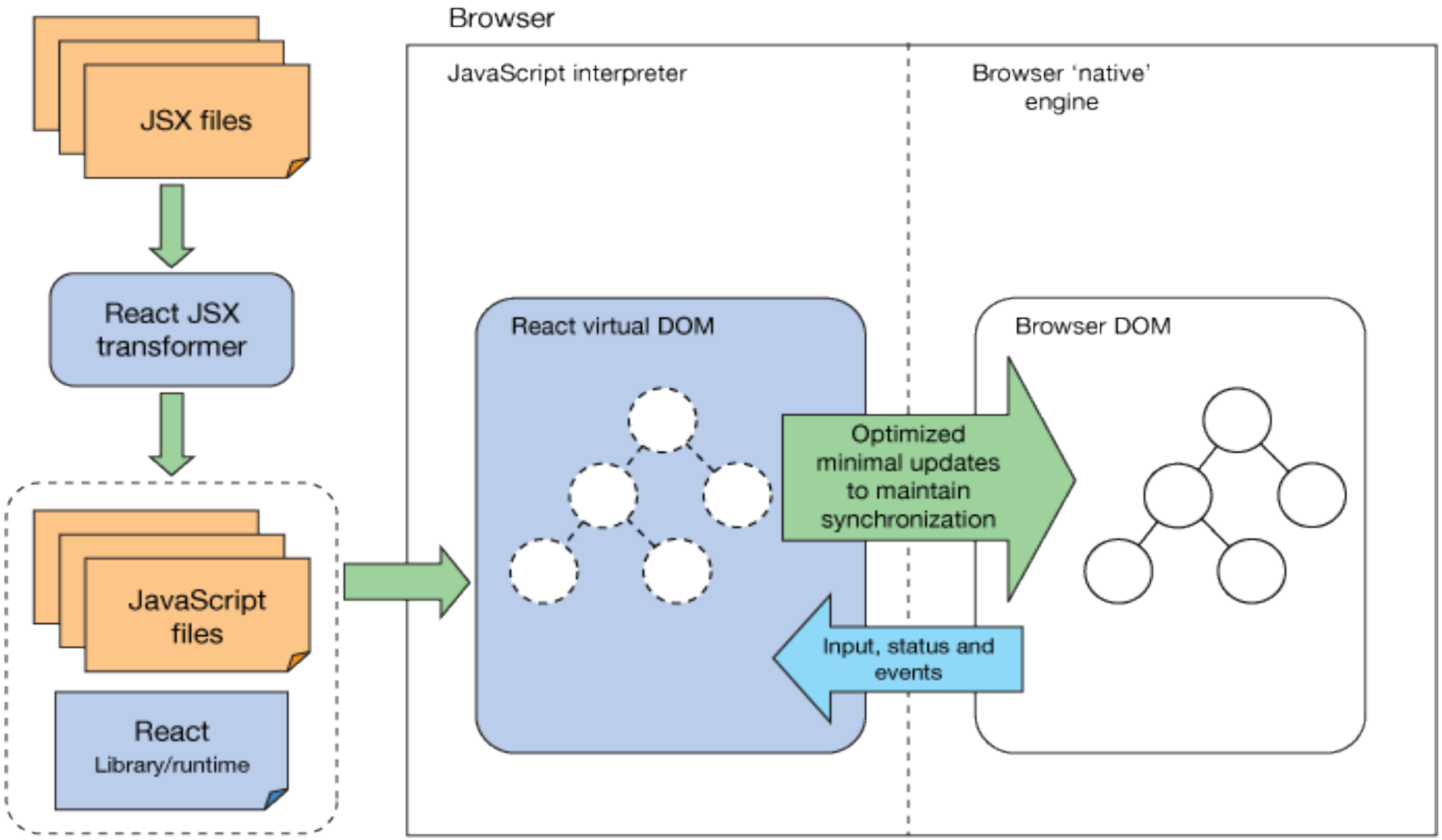
React event handlers are written inside curly braces:

`onClick={shoot}` instead of `onClick="shoot()"`.

Event Example

```
import React from 'react';
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      data: 'Initial data...'
    }
    this.updateState = this.updateState.bind(this);
  }
  updateState() {
    this.setState({data: 'Data updated from the child component...'})
  }
  render() {
    return (
      <div>
        <Content myDataProp = {this.state.data}
          updateStateProp = {this.updateState}></Content>
      </div>
    );
  }
}
class Content extends React.Component {
  render() {
    return (
      <div>
        <button onClick = {this.props.updateStateProp}>CLICK</button>
        <h3>{this.props.myDataProp}</h3>
      </div>
    );
  }
}
export default App;
```

React Virtual DOM



ReactJS Component

- A **React Component** is one of the core building blocks of React Apps.
- In other words, we can say that every application you will develop in React will be made up of pieces called components.
- Components make the task of building UIs much easier.
- Each component returns/renders some JSX code and it defines which HTML code React should render to the real DOM in the end.
- In React we mainly have two types of components: **Functional Components** and **Class Components**

Function based Component

- Functional components are JavaScript functions.
- By writing a JavaScript function, we can create a functional component in React Apps.
- To make React app efficient, we use functional component only when we are sure that our component does not require to interact with any other components.
- It’s just a function which may accepts props and returns a React element.
- But you can also use the ES6 class syntax to write components.

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

Class based Component

- The class components are similar to the functional component but has some additional features that makes class component a little more complex than the functional components.
- The functional components do not care about the other components in your app whereas the class components can work with each other.
- We can pass data from one class component to other class component.

```
class Welcome extends React.Component
{
    render()
    {
        return <h1>Hello, {this.props.name}</h1>;
    }
}
```

Differences between functional and class-Components

- The most obvious one difference is the syntax. A functional component is just a plain JavaScript function which accepts props as an argument and returns a React element.

- A class component requires you to extend from **React.Component** and create a render function which returns a React element. This requires more code but will also give you some benefits which you will see later on.

Web Reference Link

Differences between functional and class-Components

State

- Because a functional component is just a plain JavaScript function, you cannot use setState() in your component. That's the reason why they also get **called functional stateless components**. So every time you see a functional component you can be sure that this particular component doesn't have its own state.

Lifecycle Hooks

- Another feature which you cannot use in functional components are lifecycle hooks. The reason is the same like for state, all lifecycle hooks are coming from the React.Component which you extend from in class components.

Why should I use functional components at all?

- Functional component are much **easier to read and test** because they are plain JavaScript functions without state or lifecycle-hooks
- You end up with **less code**
- They help you to use **best practices**. It will get easier to separate container and presentational components because you need to think more about your component's state if you don't have access to setState() in your component
- The React team mentioned that there may be a **performance** boost for functional component in future React versions

Thank You