CS732 / SE750

Module One: Intro to React

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module One

Introduction to React

What is React?

React is a “*declarative, component-based JavaScript library for building user interfaces*”

- reactjs.org

What is React?

- **Declarative:** Allows developers to design simple views; React will take care of updating and rendering
- **Component-based:** *Separation of concerns* by building and aggregating encapsulated components which manage their own state

Motivation & benefits

1. Declarative components make for **well-defined**, **deterministic** user interfaces
2. Component-based architecture inherently promotes **code reuse** and **testability**
3. **State management** prevents malicious or accidental changes to application state
4. DOM manipulation is *slow*! React makes **fast** changes to its own *virtual DOM*, then applies the minimum possible updates to the real DOM.

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING

CS 732 / SE 750 – Module One

A simple React app

The code – our HTML file

```
<html>

<head>
    <meta charset="UTF-8">
    <title>My first React app</title>
    <script src="https://unpkg.com/react@16/umd/react.development.js" crossorigin></script>
    <script
       src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" crossorigin></script>
</head>

<body>
    <div id="container"></div>

    <script src="like-button.js"></script>
</body>

</html>
```

These `<script>`s add React to our page

This `<div>` will hold the content generated by React

This `<script>` is our own code, on the next slide…

The code – `like-button.js`

```
function LikeButton() {

    const [liked, setLiked] = React.useState(false);

    return React.createElement(
        'button',
        { onClick: () => setLiked(!liked) },
        liked ? 'Unlike' : 'Like'
    )
}
```

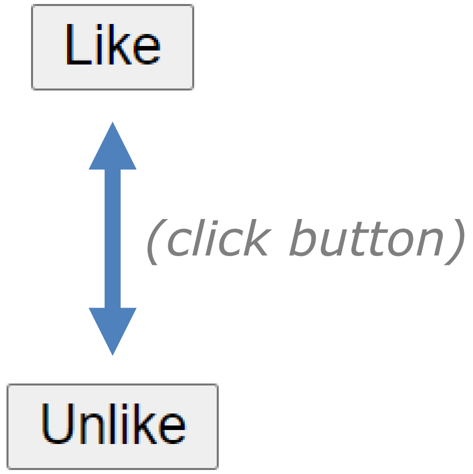
Define a React component called `LikeButton` by creating a JavaScript function like so. This component maintains a "`liked`" state, along with the ability to toggle that state by clicking it.

Note: We'll go through all the intricacies of this code in this lecture!

Find the `#container` `<div>` (see previous slide) and render a `LikeButton` inside it.

```
const container = document.querySelector("#container");
ReactDOM.render(React.createElement(LikeButton), container);
```

The result



CS 732 / SE 750 – Module One

JSX

JSX

- The previous code works well
  - However: Complex interfaces written this way will require extensive use of the `React.createElement()` function.
  - Reduces readability & maintainability when chained together
- Web developers already know a declarative language for describing user interfaces: **HTML**!
- **JSX** lets us declare our UI from within our JavaScript code, using a similar mark-up syntax

JSX

Compare this…

```
function LikeButton() {

    const [liked, setLiked] = React.useState(false);

    return React.createElement(
        'button',
        { onClick: () => setLiked(!liked) },
        liked ? 'Unlike' : 'Like'
    );
}
```

```
const container = document.querySelector("#container");
ReactDOM.render(React.createElement(LikeButton), container);
```

JSX

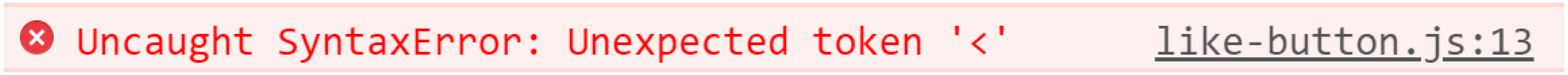
Compare this…

```
function LikeButton() {

    const [liked, setLiked] = React.useState(false);

    return (
        <button onClick={() => setLiked(!liked)}>
            {liked ? 'Unlike' : 'Like'}
        </button>
    );
}
```

```
const container = document.querySelector("#container");
ReactDOM.render(<LikeButton />, container);
```

Program output

- The code on the previous slide is syntactically correct. However:
  - Our button does not render
  - We get this error in the browser console:



```
Uncaught SyntaxError: Unexpected token '<'        like-button.js:13
```

- This is because current browsers don't support JSX syntax by default
  - We need to add *another* JavaScript library to enable this support!

Npm

- Node Package Manager (npm):
  - Default package manager that comes bundled with node.js.
  - Package managers take a config file containing a list of dependencies – in this case, package.json
  - They then take care of finding and downloading those dependencies for you – along with dependencies of dependencies!
- Npm comes bundled with Node.js.
  - Download and install the latest LTS release from: https://nodejs.org/en/
  - All platforms supported (iOS, Windows, Linux…)
  - Once installed, check the command line using **`node --version`** and **`npm --version`** – the software's version numbers should be displayed.

Babel

- Babel is known as a **“JavaScript compiler”**
  - Doesn’t “compile” in the traditional sense (no machine code / bytecode is produced)
  - More accurately described as a **transpiler** – a tool which converts *source code* written in one language, into source code written in another language.
- Originally designed to convert “modern” JavaScript into a version compatible with older browsers
- Fully customizable via plugins
  - The React dev team has created a Babel plugin for JSX!

Babel – Setup & Usage

1. Make sure you have node.js / `npm` installed
2. Install Babel in your project folder using `npm`:

```
npm init -y
npm install babel-cli@6 babel-preset-react-app@3
```

3. Create a folder to store all your JS code (e.g. `src`)
4. Start the Babel pre-processor

```
npx babel --watch src --out-dir . --presets react-app/prod
```

Starts Babel and instructs it to watch all files here…

… When a file in the `src` directory changes, a processed version will be created here.

5. Have your HTML file reference the generated file in a `<script>` tag - everything should work fine! You can check out the generated file in a text editor (or viewing source in the browser) to see what Babel produced.

ENGINEERING

CS 732 / SE 750 – Module One

React toolchains

Toolchains

- A toolchain is a set of tools used to perform a complex software development task or otherwise aid software developers
  - Other examples?
- Use of Babel to allow JSX syntax is an example of a simple toolchain to aid React developers
- Web developers commonly use more complex toolchains to provide additional functionality

React toolchains

- Popular toolchains for React developers include:
  - Vite+React
  - Create React App (CRA)
  - Next.js
- We will examine Vite+React and Next.js in this course, but all of these provide extensive benefits in terms of development and application optimization

Vite

- https://vitejs.dev/guide/
- A **fast**, **powerful**, **modern** toolkit for creating JavaScript-based frontends in a variety of frameworks
  - Including **React**, Vue, Svelte…
- Key features:
  - Dependency resolution
  - Hot module replacement
  - TypeScript*
  - CSS, PostCSS, CSS modules

Vite – Getting started

- To get started, simply run the following command, then follow the prompts:

```
npm create vite@latest
```

- Options:
  - **Project name:** Your project will be created in a folder with this name (e.g. my-first-vite-project)
  - **Select a framework:** You'll probably be picking "React" here!
  - **Select a variant:** JS or TypeScript, your choice (examples in this course use JS). SWC is an experimental **FAST** compiler – works really well in my testing (examples use this)!

Project structure

- **public:** Files in here can be directly navigated to by the browser
- **src:** Contains your source code **and CSS.**
  - *.jsx / *.tsx: Your React code. Anything using JSX needs to be placed in a file with this extension (i.e. *.js or *.ts wont work if there's any JSX code).
  - main.jsx / main.tsx: Program entry point.
- **.gitignore:** Prevents certain unnecessary files from being added to your git repos.
- **index.html:** The "base" page for your app. React components will be "injected" into the "root" div.
- **package.json:** Contains project dependencies
- **vite.config.js:** Contains vite config. Probably don't need to touch this (yet!)

App development

- After setup, navigate to the project directory
- Before you run *for the first time*, install dependencies:

```
npm install
```

- Then, start the dev environment:

```
npm run dev
```

- You'll see something like this:

```
VITE v4.1.2  ready in 442 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  press h to show help
```

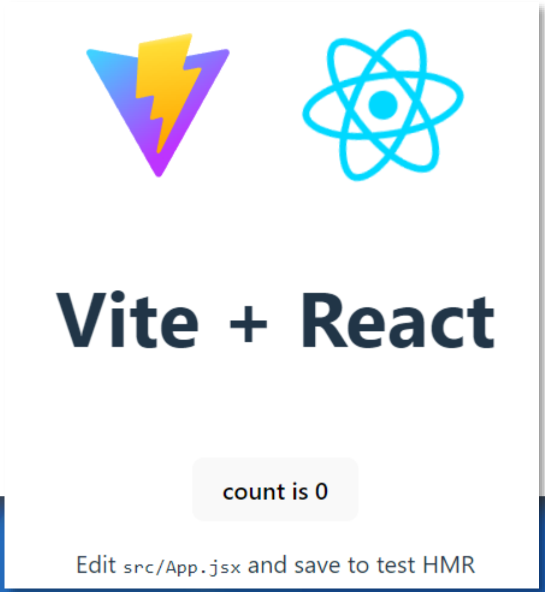
Browse here to view your webapp (link is ctrl+clickable in most terminals)

App development

- Now, let's test Hot Module Reloading:
  - Go to App.js, and make any change, e.g:

```
<h1>~~Vite + React~~Hello World!</h1>
```

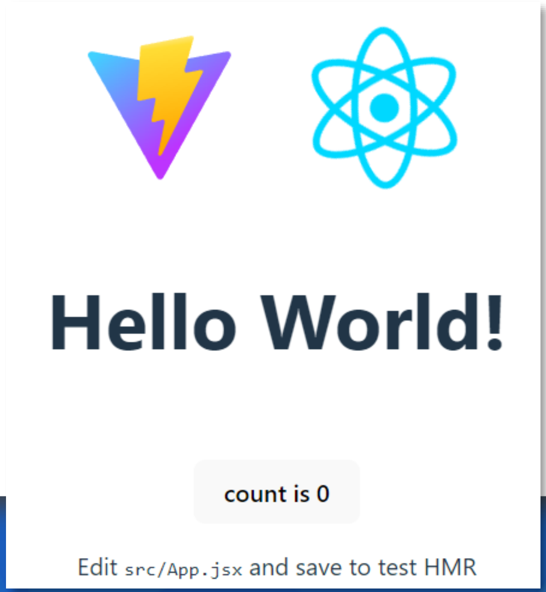
  - Once you save your changes, the changes should *immediately* be visible in the browser – no refresh required!



Vite + React

count is 0

Edit `src/App.jsx` and save to test HMR



Hello World!

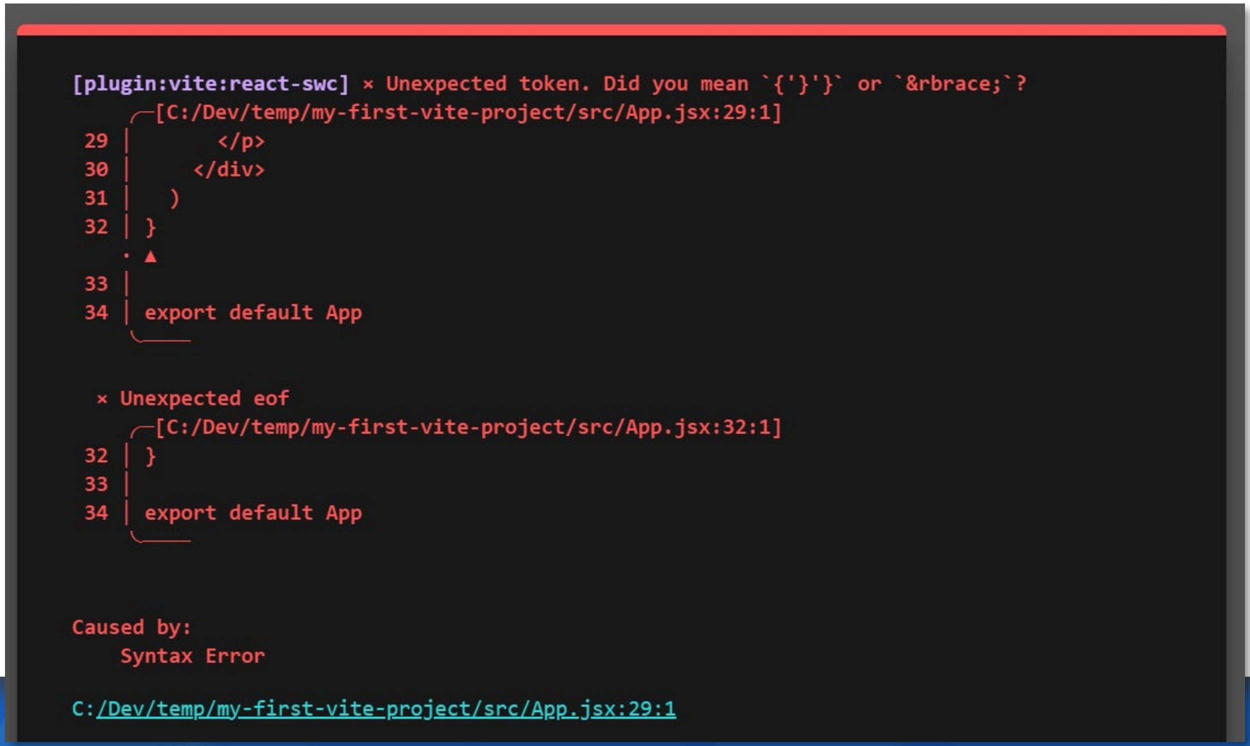
count is 0

Edit `src/App.jsx` and save to test HMR

App development

- If you make a syntax error, the browser window should tell you what it is! E.g:

`<h1>~~Vite + React~~Hello World!` *(missing </h1>)*



```
[plugin:vite:react-swc] × Unexpected token. Did you mean `{'}'}` or `&rbrace;`?
   ╭─[C:/Dev/temp/my-first-vite-project/src/App.jsx:29:1]
29 │         </p>
30 │       </div>
31 │     )
32 │ }
   · ▲
33 │
34 │ export default App
   ╰────

  × Unexpected eof
   ╭─[C:/Dev/temp/my-first-vite-project/src/App.jsx:32:1]
32 │ }
33 │
34 │ export default App
   ╰────


Caused by:
    Syntax Error

C:/Dev/temp/my-first-vite-project/src/App.jsx:29:1
```

CS 732 / SE 750 – Module One

React components

Components

- React apps are built from reusable components
- Components may have their own state, properties & style
- Typically (but not required) components will be written in their own file for readability / separation

Functional components

- Functional components are thus-called as they are written as a single JavaScript function, which may optionally take a single argument for its properties

```
function App() {
  return (
    <div>
      <p>Hello, world!</p>
    </div>
  );
}
```

Defines a functional component, called `App`. This component is rendered as a `<div>` containing a single `<p>` with the text "Hello, world!".

```
export default App;
```

Exports the `App` function so it can be used outside this file (with a corresponding `import`).

Functional components

- These can also be written using arrow function syntax (lambdas)
  - Can be “shorthand” for simple components such as this
  - For more complex components there is little difference – the choice is a matter of preference

```
const App = () =>
  <div>
     <p>Hello, world!</p>
  </div>

export default App;
```

Class components

- Class components are written as ES6 JavaScript classes. They can:
  - Maintain their own state
  - Hook into the React *component lifecycle* (i.e. be notified when they will be mounted / unmounted / etc).

```
import React from 'react';

class MyFirstComponent extends React.Component {

    constructor(props) {
        super(props);
        // TODO Initialize any state here
    }

    render() {
        return (
            <p>This component is awesome!!</p>
        );
    }
}

export default MyFirstComponent;
```

ES6 class syntax – very similar to Java

We may override the constructor if we need to perform some initialization. Otherwise, we can leave this part out.

The `render()` function returns how this component should be rendered in the browser

Class components

- Class components are written as ES6 JavaScript classes. They can:
  - Maintain their own state
  - Hook into the React *component lifecycle* (i.e. be notified when they will be mounted / unmounted / etc).

```
import React from 'react';

class MyFirstComponent extends React.Component {

    constructor(props) {
        super(props);
        // TODO initialize any state here
    }

    render() {
        return (
            <p>This component is awesome!!</p>
        );
    }
}

export default MyFirstComponent;
```

Note: In previous versions of React, you needed to write Class components if those components needed to use lifecycle methods and / or maintain state. Now, with React hooks, you no longer need to do this. Therefore, we will focus on functional components in this course.

Using components

- You can use components within other components

```
import React from 'react';
import MyFirstComponent from './my-first-component';

function App() {
  return (
    <div>
      <MyFirstComponent />
    </div>
  );
}

export default App;
```

Import `MyFirstComponent`, which may be a `function` or a `class` – the syntax works either way. The component is imported from `./my-first-component.js`

Use it, just as with `<div>`, `<p>` or any other built-in element

CS 732 / SE 750 – Module One

Component properties

Properties

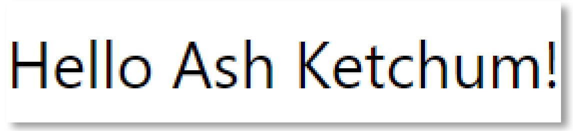
- Properties allow parent components to pass configuration down to children
- Using JSX, properties are assigned using standard XML-like markup `attribute=value` syntax
- For example:

```
<Greeting firstName="Ash" lastName="Ketchum" />
```

Properties

- Passed properties are accessible via the function argument

```
function Greeting(props) {
    return (
        <p>Hello {props.firstName} {props.lastName}!</p>
    );
}
```


Hello Ash Ketchum!

- Can use object destructuring to avoid having to write “props” everywhere in the function if desired:

```
function Greeting({firstName, lastName}) {
    return (
        <p>Hello {firstName} {lastName}!</p>
    );
}
```

{ } syntax

- The { } syntax seen on the previous slide can contain any JavaScript **expression**.
- The expression is evaluated, and its value is substituted in the {}.
- On the previous slide, the value is a string – but it could be *anything* – even more JSX!

CS 732 / SE 750 – Module One

Logic in JSX

Boolean logic

- We can have *control flow* such as *conditionals* and *loops* within our render logic – but sometimes the syntax may be different than you're used to.
- Simple conditional example: Create a component called `AgeCheck`, with the following requirements:
  - A single property: `age`
  - If age is >= 18, the component will render a `<p>` with the message "You're 18 or older".
  - Otherwise, a `<p>` with the message "You're not old enough to see this" will be rendered.
- How could we do this?

Boolean logic

(we can put "export default" here as a shortcut instead of having it on a separate line later)

```
export default function AgeCheck(props) {

    if (props.age >= 18) {

        return <p>You're 18 or older</p>
    }
    else {

        return <p>You're not old enough to see this</p>
    }
}
```

Standard `if-else` syntax – we return a different `<p>` depending on the value of `props.age`

Boolean logic

- More complex example: let's develop a component named `BusinessCard`, with the following requirements:
  - Takes four optional properties: `name`, `company`, `phoneNum`, and `email`.
  - Each of these properties is rendered in a `<table>`, for example:

```
<table>
    <tr><th>Name:</th><td>Ash Ketchum</td></tr>
    <tr><th>Company:</th><td>Silph Co.</td></tr>
    <tr><th>Ph #:</th><td>00 123 4567</td></tr>
    <tr><th>Email:</th><td>ash@silphco.biz</td></tr>
</table>
```

  - **If a property isn't supplied, that `<tr>` won't be rendered at all.**
- How could we do this?

Boolean logic

- We can’t use `if`-statements inside JSX - but we *can* use the ternary operator!
- The ternary operator evaluates a given expression *as a Boolean*, and returns one value (before the **:**) if `true`, and another value (after the **:**) if `false`.
  - In JavaScript, the values “”, `0`, `null`, `NaN`, `undefined` and `false` evaluate to `false`.
  - This means we can conditionally render some elements based on whether a given property was supplied (i.e. its value isn’t `undefined`).

```
export default function BusinessCard(props) {
    return (
        <table>
            {props.name ? <tr><th>Name:</th><td>{props.name}</td></tr> : null}
            {props.company ? <tr><th>Company:</th><td>{props.company}</td></tr> : null}
            {props.phoneNum ? <tr><th>Ph #:</th><td>{props.phoneNum}</td></tr> : null}
            {props.email ? <tr><th>Email:</th><td>{props.email}</td></tr> : null}
        </table>
    );
}
```

Condition

Value if `true`

Value if `false` (`null` will not be rendered)

Boolean logic

- If we only want to render something when a value is “true” (or exists), we can use && syntax as a further shortcut
- This syntax will evaluate the given Boolean expression, and will only return the value after the && if the expression evaluates to “true”.
- This is the preferred syntax when we don’t need to return something *or* something else.

```
export default function BusinessCard(props) {
    return (
        <table>
            {props.name && <tr><th>Name:</th><td>{props.name}</td></tr>}
            {props.company && <tr><th>Company:</th><td>{props.company}</td></tr>}
            {props.phoneNum && <tr><th>Ph #:</th><td>{props.phoneNum}</td></tr>}
            {props.email && <tr><th>Email:</th><td>{props.email}</td></tr>}
        </table>
    );
}
```

Condition

Value if true

Iteration

- Similarly to conditionals, we cannot write loops within JSX
- However, we can output arrays within {}

```
{[<p>1</p>, <p>2</p>, <p>3</p>]}
```

- We can use this, along with the JavaScript array's `map()` function
  - This converts an array of elements of some type into an array of elements of another type – for example, an array of strings into an array of <p>'s

```
{["a", "b", "c"].map((item, index) => <p key={index}>{item}</p>)}
```

`map()` calls this function once for each element in the source array, to generate the result array elements. Each element in the generated array should have a unique key.

ENGINEERING

CS 732 / SE 750 – Module One

Styling React components

Styles

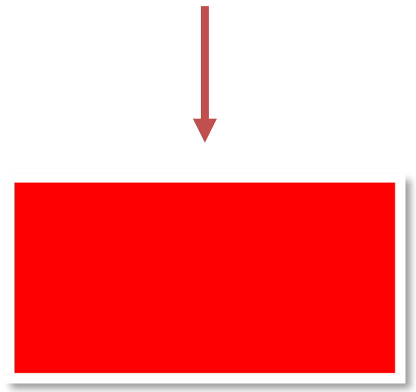
- Three ways to add CSS styles to our components:
  - Inline styles
  - Standard CSS import
  - CSS modules

Inline styles

- Components have a `style` property which can be set
- Any CSS property can be set this way
  - Note the camel case (e.g. `backgroundColor` instead of `background-color`)

```
export default function Square(props) {

    const boxStyle = {
        width: props.width,
        height: props.height,
        backgroundColor: props.backgroundColor,
        margin: '5px'
    };

    return <div style={boxStyle} />
}
```

```
<Square width="200px" height="100px"
   backgroundColor="red" />
```



Standard CSS import

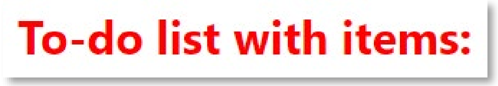
- We can use the `import` statement to import CSS files as-is in addition to JS files

```
import './index.css';
```

- Webpack will take care of packaging up any referenced CSS files and making their contents available to the browser
- Any so-imported CSS rules are *global* – they apply to your entire site.
  - Best-practice to import such CSS files from within `index.js` to signify this
- You can assign CSS classes to components – use the `className` property rather than the `class` property.

```
.important {
  color: red;
  font-weight: bold;
}
```

```
<p className="important">To-do list with items:</p>
```



To-do list with items:

CSS modules

- CSS modules allow CSS to be applied locally, to specific components. To use:
  1. Name the CSS file *.module.css (e.g. `ArticleView.module.css`)
  2. Use an import of the following form, within a component JS file:

     ```
     import styles from './ArticleView.module.css';
     ```

  3. Assign class names to elements like so:

     ```
     <div className={styles.article}> … </div>
     ```

- PostCSS will dynamically modify the CSS class selectors supplied to the browser, to avoid naming conflicts (e.g. "`.box`" in one module file won't conflict with "`.box`" in another module file).

  ```
  .article { … }  ⟶  .ArticleView_article__hnmoW { … }
  ```

- **Note:** All CSS selectors in module files *must start with* a class selector *(why?)*

CS 732 / SE 750 – Module One

React hooks

React hooks

- New addition to React (version 16.8 onwards)
- In prior versions, needed to write *class* components to give components state and to access lifecycle methods (i.e. being notified when components are mounted / unmounted / rendered)
- Now, we can add the same functionality to functional components using hooks
- We can also write custom hooks to handle more advanced logic and share stateful rendering code (more on this in later weeks!)

CS 732 / SE 750 – Module One

Component state with useState()

State

- A component’s properties should be considered **immutable** – they don’t change once set.
- Components may maintain local state, which *can* change.
- For class-based components:
  - A variable (`this.state`) plus a method to update the state (`this.setState()`). You may read more on this here.
- For functional components:
  - From version 16.8 onwards, we can use the React hook, `useState()`.

State with useState()

- To use the function, we must import it:

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

- Next, we can use it as follows:

```
function LikeButton() {

    const [liked, setLiked] = useState(false);

    return (
        <button onClick={() => setLiked(!liked)}>
            {liked ? 'Unlike' : 'Like'}
        </button>
    );
}
```

useState() – A closer look

This will be an *array* with **two** elements:

- Index [0] will be the value itself
- Index [1] will be a function we can call to change the value

This value is the initial value for the state (in this case, the Boolean value “false”). It’s only used the first time this is rendered.

```
const likedState = useState(false);

const liked = likedState[0];
const setLiked = likedState[1];
```

useState() – A closer look

It is much more common (and **better practice!**) to define the value and mutator function using array dereferencing as follows.

```
~~const likedState = useState(false);~~

~~const liked = likedState[0];~~
~~const setLiked = likedState[1];~~
```

```
const [liked, setLiked] = useState(false);
```

useState() – A closer look

```
function LikeButton() {

    const [liked, setLiked] = useState(false);

    return (
        <button onClick={() => setLiked(!liked)}>
            {liked ? 'Unlike' : 'Like'}
        </button>
    );
}
```

And we can change the value by calling the mutator function as so.

Calling this function will cause the component to be re-rendered, thus showing the updated state to the user.

We can use the value as if it were any other variable / property

ENGINEERING

CS 732 / SE 750 – Module One

Side-effects with `useEffect()`

useEffect() hook

- Sometimes, in a React component, we want to do things other than just render the component.
- Examples:
  - Manually update non-React parts of the DOM
  - Fetch data
  - Subscriptions
  - Timers
- In class-based components, we can perform these actions in the lifecycle methods
- In functional components, we can use the `useEffect()` hook.

Effects with no cleanup

- To use `useEffect()`, we can supply a single argument – a function, which will be called after the component has finished rendering.
- The below example will update the document title (which appears on the browser tab) after every render:

```
useEffect(() => {
    document.title = `Counter value: ${value}`;
});
```

Effects with cleanup

- Sometimes we need to clean up after ourselves. For example:
  - Stopping timers that we've started
  - Unsubscribing from APIs we've subscribed to
- To do this we can have the effect function return another function
  - This second function will be called just before the component unmounts, or before the effect function is called again.
- Sometimes, we *don't* want to call the effect function again under certain circumstances.
  - To do this, we can supply an optional second argument. Your function will only be called again if the values supplied in the second argument have changed since the last time it was called.

useEffect() example

- An example, showing how timers can be started / stopped:

```
const [seconds, setSeconds] = useState(0);
const [isActive, setActive] = useState(false);

useEffect(() => {
    let interval = null;
    if (isActive) {
        interval = setTimeout(() => setSeconds(seconds + 1), 1000);
    }

    return () => clearTimeout(interval);

}, [seconds, isActive]);
```

The effect function sets a timeout – a function to be called after a 1000ms delay.

Before the component is unmounted or re-rendered, any pending timeout is cleared.

We only want to re-call the effect function again if the timer-related values supplied here have changed.

CS 732 / SE 750 – Module One

useRef()

useRef() hook

- Lets you reference a value which is not needed for rendering purposes
- Most commonly used to get a reference to a React / HTML element
- useRef • React (reactjs.org)

Practical usage example: HTML forms

- Form handling with just useState():
  - Known as "bound forms"

```
export default function FormWithUseState() {

    const [name, setName] = useState('');

    const handleClick = () => alert(`My name is ${name}`);

    return (
        <>
            <form>
                <label htmlFor="txtForm1">Name:</label>
                <input id="txtForm1" type="text" value={name} onChange={e => setName(e.target.value)} />
            </form>
            <button onClick={handleClick}>What's my name?</button>
        </>
    )
}
```

Practical usage example: HTML forms

- And with useRef():

```
export default function FormWithUseRef() {
    const inputRef = useRef(null);
    const handleClick = () => alert(`My name is ${inputRef.current.value}`);
    return (
        <>
            <form>
                <label htmlFor="txtForm2">Name:</label>
                <input ref={inputRef} id="txtForm2" type="text" />
            </form>

            <button onClick={handleClick}>What's my name?</button>
        </>
    )
}
```

Tradeoffs

useState() approach

- **Pro:** Other UI elements using the same stateful value will be updated immediately as the user types
- **Con:** If we don't need the insta-update functionality, then we're just causing unnecessary re-renders
  - Hurts performance

useRef() approach

- **Pro:** No unnecessary re-renders
  - Better performance
- **Con:** Can't have other UI elements update immediately as the user types