ReactJS Part 1

IMY 220 • Lecture 17

What is React?

It was created and is maintained by Facebook

“React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It lets you compose complex UIs from small and isolated pieces of code called ‘components’.”

https://reactjs.org/tutorial/tutorial.html

React is a front-end framework

Why React?

React has a number of strengths which make it useful as a front-end JS framework

We will focus on two of these:

- The Virtual DOM
- JSX

The Virtual DOM

When we use something like AJAX to manage information and elements on a page, we are interacting with the DOM API

- For example, when we use document.createElement or document.appendChild

Some of these DOM instructions are very slow, which means that if we're constantly creating and destroying elements, we are paying a huge performance cost

The Virtual DOM

When using React, we don’t interact with the DOM directly

Instead, we interact with the *Virtual* DOM which consists out of JS objects called React elements

> (Interacting with JS objects is much faster performance-wise that interacting with the DOM)

React then uses the virtual DOM to efficiently manage changes to the (real) DOM

https://reactjs.org/docs/faq-internals.html#what-is-the-virtual-dom

The Virtual DOM

“A React element is a description of what the actual DOM element should look like. In other words, React elements are the instructions for how the browser DOM should be created.”

Learning React

So, generally speaking, we use React to create React elements, which can consist of other React elements, DOM elements and text nodes, and then render them to the DOM

React elements can be created in many ways…

React Elements

```
const greeting = React.createElement("div", {"className": "container"},
   React.createElement("h1", null, "Hello there!")
);
```

The example above creates a React element (greeting) that consists of a div with a header (with a text node) nested inside

The React.createElement function is used to create an element which can be rendered to the DOM. React.createElement is the basis of much of React's functionality

React Elements

In other words, this…

```
const greeting = React.createElement("div", {"className": "container"},
    React.createElement("h1", null, "Hello there!")
);
```

…creates a virtual DOM version of this…

```
<div class="container">
    <h1> Hello there! </h1>
</div>
```

React Elements

```
const GreetingBox = React.createClass({
    displayName: "Greeting",
    render(){
        return React.createElement("div", {"className": "container"},
            React.createElement("h1", null, "Hello there!")
        )
    }
});
const greeting = React.createElement(GreetingBox, null, null);
```

This example does the same thing, but first defines GreetingBox as a **component** (we'll deal with components in a while)

React Elements

```
const GreetingBox = ({name}) =>
    React.createElement("div", {"className": "container"},
        React.createElement("h1", null, "Hello " + name)
    );

const greeting = React.createElement(GreetingBox, {"name": "Diffie"}, null);
```

This example creates a stateless functional component

It also has a “name” property. (Just like DOM elements, React elements can have properties. These can be used to alter their behaviour.)

React Elements

However, in this module we'll focus on using a React-specific syntax extension called JSX (**J**ava**S**cript **X**ML) to create React elements

JSX allows you to create React elements using a combination of JS and an XML-syntax, for example

```
const greeting = (
   <div>
      <h1>
         Hello React!
      </h1>
   </div>
);
```

Including React (+ some issues)

But first… let’s look at including and using React

As usual, you can download and include the JS files or link to CDN versions. React is split up into two packages called React and ReactDOM

- This allows core React functionality to be used efficiently for non-browser contexts, such as mobile applications

Since we are going to use React exclusively in the browser, we need to include both packages

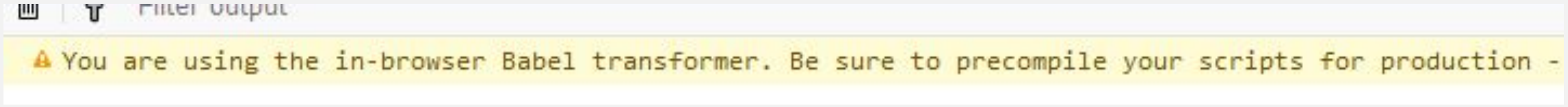
Including React (+ some issues)

```
<script crossorigin
src="https://unpkg.com/react@16/umd/react.production.min.js"></script>
<script crossorigin
src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js"></scri
pt>
```

However, if we want to use JSX (which is obviously not valid JS syntax) we have to include and use Babel

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.26.0/babel.j
s"></script>
<script type="text/babel">
   // All your JS goes here
</script>
```

Including React (+ some issues)



```
Filter output
⚠ You are using the in-browser Babel transformer. Be sure to precompile your scripts for production -
```

NB!!! You should never include and use Babel as shown for production websites

For production websites, you need to use a tool like Webpack that has a *loader* which converts JSX code into something that the browser can understand

However, for these two lectures, we’re going to use the in-browser Babel transformer as shown in the previous slide

React Elements

In order to view React elements in the browser, we have to **render** them using a method called ReactDOM.render

This method is part of the ReactDOM package and takes two parameters:

- A React element to be rendered
- An HTML element to render the React element to

```
// Using any one of our previously created "greeting" React elements
// ...as well as an (HTML) element with an id of "react-container"
ReactDOM.render(
    greeting,
    document.getElementById("react-container")
);
```

React Elements

In order to view React elements in the browser, we have to render them using a method called createRoot

Rendering HTML to the browser page involves two steps:

- Selecting the root HTML element to render React elements to using createRoot
- Rendering the React elements using render

```
// Using any one of our previously created "greeting" React elements
// ...as well as an (HTML) element with an id of "root"
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<greeting />);
```

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


<script crossorigin
src="https://unpkg.com/react@18/umd/react.development.js"></script>
<script crossorigin
src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>
<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>


<script type="text/babel">
    const greeting = (
        <div>
            <h1> Hello React! </h1>
        </div>
    );


    const root = ReactDOM.createRoot(document.getElementById("root"));
        root.render(<greeting />);
</script>
```

React Components

However, we will focus on using React **Components** in this module

“Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen.”

https://reactjs.org/docs/components-and-props.html

Components can also be defined in a variety of ways. We will focus on using ES6 classes to define components.

```jsx
class Greeting extends React.Component{
   render(){
      return (
         <div>
             <h1>
                Hello React!
             </h1>
         </div>
      );
   }
}

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<Greeting />);
```

React Components - example

Let’s go through this example bit by bit and discuss how it works

```
class Greeting extends React.Component{
```

In order to create our own component, we need to create our own subclass which inherits from the React.Component class

Class names (and thus component names) should always be capitalised, as per convention

React Components - example

```
render(){
   return (
      <div>
         <h1>
            Hello React!
         </h1>
      </div>
   );
}
```

nat components *return*
nder the result
define a render()
returns a react element.

The render() function is required when defining a component
(All other functions are optional)

React Components - example

```
ReactDOM.render(
   <Greeting />,
   document.getElementById("react-container")
);
```

Now that we've defined our component, we can render it using the ReactDOM.render() method. This calls the render() method we defined earlier in our component

Note that we **have** to close elements when using React. Using a JSX element without a closing tag, for example <Greeting>, will cause an error

React Components - example

```
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<Greeting />);
```

Now that we've defined our component, we can render it using the render() method. This calls the render() method we defined earlier in our component

Note that we **have** to close elements when using React. Using a JSX element without a closing tag, for example <Greeting>, will cause an error

React Components

We can also inject JS into JSX by enclosing it in curly brackets, for example:

```
let name = "Diffie";
class Greeting extends React.Component {
    render(){
        return (
            <h1> {"Hello " + name} </h1>
        );
    }
}
```

The JS between curly braces is evaluated and the values are returned

React Components

We can also inject JS into JSX by enclosing it in curly brackets, for example:

```
let name = "Diffie";
class Greeting extends React.Component {
    render(){
        return (
            <h1> {"Hello " + name} </h1>
        );
    }
}
```

Note: this is purely for illustrative purposes. You should **never** have to use a global variable inside a component like this

The JS between curly braces is evaluated and the values are returned

React Components

Components can also contain other components

This is a big part of the strength of React: creating reusable pieces of code that represent small pieces of content and/or functionality

This helps keep our code clear and scalable

```
class ListItem extends React.Component {
   render(){
      return (
         <li> My custom list item </li>
      );
   }
}

class List extends React.Component {
   render(){
      return (
         <ul>
            <ListItem />
            <ListItem />
            <ListItem />
         </ul>
      );
   }
}
```

React Components

You can then render List as usual

```
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<List />);
```

Using components inside one another allows us to abstract the creation of elements and reuse this abstraction to create more complex content and/or functionality in an intuitive, readable way

React Components

Note that you cannot return multi
elements without wrapping them
inside a parent element, for
example, instead of doing this…

```
render(){
   return (
      <h1> Hello React! </h1>
      <p> This is not going to work </p>
   );
}
```

…you have had to do this:

(the first will give you an error)

```
render(){
   return (
      <div>
         <h1> Hello React! </h1>
         <p> This will work </p>
      </div>
   );
}
```

React Components

However, wrapping elements to be rendered in a parent element, such as a div, doesn’t always make sense, for example

```
class Table extends React.Component {
  render(){
    return (
      <table>
        <tr>
          <Columns />
        </tr>
      </table>
    );
  }
}
```

```
class Columns extends React.Component {
  render(){
    return (
      <div>
        <td>Hello</td>
        <td>World</td>
      </div>
    );
  }
}
```

React Components

However, wrapping elements to be rendered in a parent element, such as a div, doesn't always make sense, for example

```
class Table extends React.Component {
  render(){
    return (
      <table>
        <tr>
          <Columns />
        </tr>
      </table>
    );
  }
}
```

```
class Columns extends React.Component {
  render(){
    return (
      <div>
        <td>Hello</td>
        <td>World</td>
      </div>
    );
  }
}
```

This is obviously not ideal

React Components

The recommended way to solve this problem is by using what is called the Fragments API, which works by wrapping elements in a parent element called <React.Fragment>

```
render(){
  return (
    <React.Fragment>
          <td>Hello</td>
          <td>World</td>
      </React.Fragment>
    );
}
```

React Components

The recommended way to solve this problem is by using what is called the Fragments API, which works by wrapping elements in a parent element called <Fragment>

```
render(){
   return (
      <>
            <td>Hello</td>
            <td>World</td>
        </>
      );
}
```

React Components

You can also add other code inside the render() function

```
class List extends React.Component {
   render(){
      let listItems = [], listLength = 5;
      for(let i = 0; i < listLength; i++){
         listItems.push( (<ListItem />) );
      }

      return (
         <ul>{listItems}</ul>
      );
   }
}
```

Note the extra brackets around the JSX

Component properties

Like HTML elements, React components can also contain “attributes”

```
<Greeting personName="Diffie" />
```

However, in React components, these are called props (short for properties) and [illegible] in this.props.(attributeName)

```
render(){
   return (
      <div>
         <h1>
            Hello {this.props.personName}!
         </h1>
      </div>
   );
}
```

```
class Greeting extends React.Component{
   render(){
      return(
         <div>
            <h1>
               Hello {this.props.personName}!
            </h1>
         </div>
      );
   }
}

ReactDOM.render(
   <Greeting personName="Diffie" />,
   document.getElementById("react-container")
);
// Output: Hello Diffie!
```

```
class Greeting extends React.Component{
   render(){
      return(
         <div>
            <h1>
               Hello {this.props.personName}!
            </h1>
         </div>
      );
   }
}

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<Greeting personName="Diffie" />);

// Output: Hello Diffie!
```

Component properties

JS variables can also be passed to a component's props

For example, if we want to render a list using an array of JS objects, we can pass it as the props of a component:

```
let peopleList1 = [
   {name: "Troy", surname: "Barnes"},
   {name: "Abed", surname: "Nadir"}
];

ReactDOM.render(
   <PersonList people={peopleList1} />,
   document.getElementById('root')
);
```

Since we are injecting JS variables into JSX, we need to enclose it in curly brackets. Similar to previous examples, any JS between the curly brackets is evaluated and the values are returned

Also note the lack of inverted commas

Component properties

JS variables can also be passed to a component's props

For example, if we want to render a list using an array of JS objects, we can pass it as the props of a component:

```
let peopleList1 = [
   {name: "Troy", surname: "Barnes"},
   {name: "Abed", surname: "Nadir"}
];


const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<PersonList people={peopleList1} />);
```

Since we are injecting JS variables into JSX, we need to enclose it in curly brackets. Similar to previous examples, any JS between the curly brackets is evaluated and the values are returned.
Also note the lack of inverted commas.

Component properties

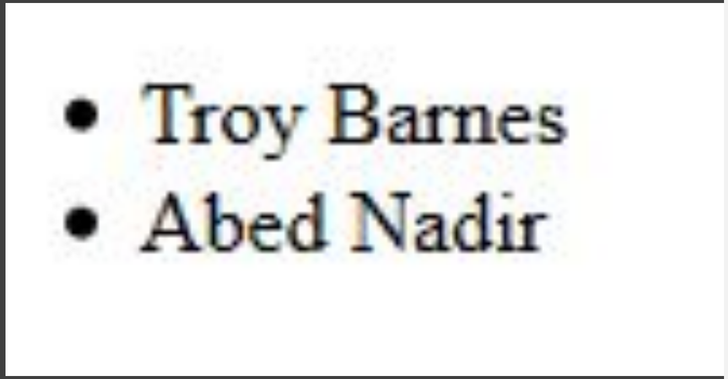
Then we can access it via this.props.(attributeName)

```
class PersonList extends React.Component {
   render(){
      return (
         <ul>
            {this.props.people.map((person, i) => {
               return <li> {person.name} {person.surname} </li>;
            })}
         </ul>
      );
   }
}
```

```
class PersonList extends React.Component {
   render(){
      return (
         <ul>
            {this.props.people.map((person, i) => {
               return <li> {person.name} {person.surname} </li>;
            })}
         </ul>
      );
   }
}

let peopleList1 = [
   {name: "Troy", surname: "Barnes"},
   {name: "Abed", surname: "Nadir"}
];

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<PersonList people={peopleList1} />);
```



- Troy Barnes
- Abed Nadir

Component properties

Finally, we can also pass variables between components using props

For example, we can create a component which encapsulates a single person in a list and prints out the first letter of their name, followed by a full stop and then their surname (for example, T. Barnes)

```
class Person extends React.Component {
   render(){
      return (
         <li>{`${this.props.person.name[0]}. ${this.props.person.surname}`}</li>
      );
   }
}
```

Component properties

We can then create a PersonList component which prints out a heading stating the number of people in the list, followed by the list itself

```
class PersonList extends React.Component {
   render(){
      return (
         <div className="container">
            <h1>
               {this.props.people.length} people in the list:
            </h1>
            <ul>
               {this.props.people.map((person, i) => {
                  return <Person key={i} person={person} />;
               })}
            </ul>
         </div>
      );
   }
}
```

Side note

There are two new things to note in the previous slide:

- The parent div is given a className attribute of “container”. Since class is a JS keyword as of ES6, we can’t set an element’s class attribute using the class keyword, so when doing React, we have to use className

- When creating list items, we should always give list elements a unique key attribute. *“Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity”*
  https://reactjs.org/docs/lists-and-keys.html

Component properties

As per the previous example, we simply pass the array of people as the props for the PersonList component, and it will use the array to create an appropriate heading and list

```
let peopleList1 = [
   {name: "Troy", surname: "Barnes"},
   {name: "Abed", surname: "Nadir"}
];

ReactDOM.render(
   <PersonList people={peopleList1} />,
   document.getElementById('root')
);
```

Component properties

As per the previous example, we simply pass the array of people as the props for the PersonList component, and it will use the array to create an appropriate heading and list

```
let peopleList1 = [
   {name: "Troy", surname: "Barnes"},
   {name: "Abed", surname: "Nadir"}
];


const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<PersonList people={peopleList1} />);
```

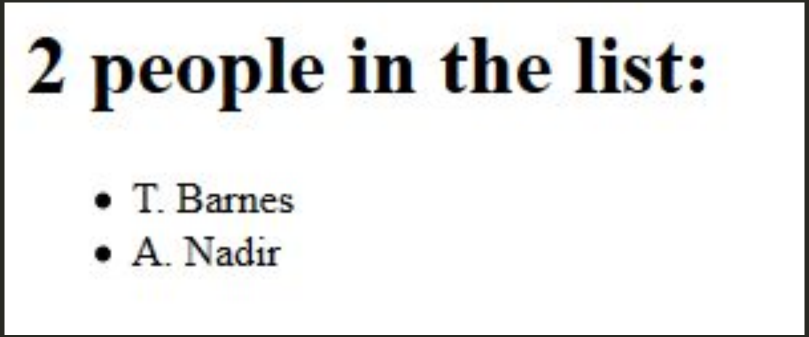
```
class Person extends React.Component {
    render(){
        return (
            <li>{`${this.props.person.name[0]}. ${this.props.person.surname}`}</li>
        );
    }
}

class PersonList extends React.Component {
    render(){
        return (
            <div className="container">
                <h1>
                    {this.props.people.length} people in the list:
                </h1>
                <ul>
                    {this.props.people.map(function(person, i){
                        return <Person key={i} person={person} />;
                    })}
                </ul>
            </div>
        );

    }
}

var peopleList1 = [
    {name: "Troy", surname: "Barnes"},
    {name: "Abed", surname: "Nadir"}
];

ReactDOM.render(
    <PersonList people={peopleList1} />,
    document.getElementById('root')
);
```



2 people in the list:

- T. Barnes
- A. Nadir

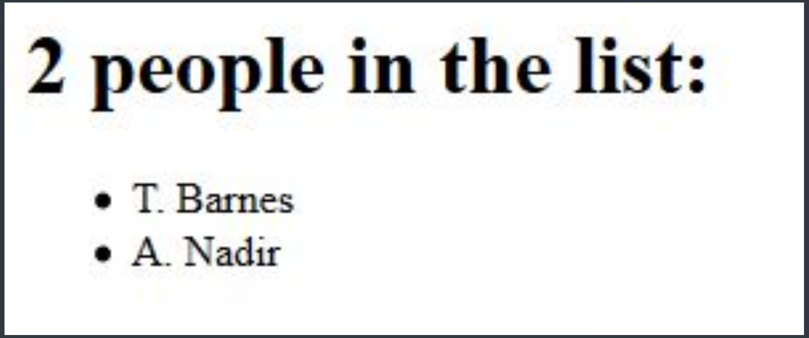
```
class Person extends React.Component {
    render(){
        return (
            <li>{`${this.props.person.name[0]}. ${this.props.person.surname}`}</li>
        );
    }
}

class PersonList extends React.Component {
    render(){
        return (
            <div className="container">
                <h1>
                    {this.props.people.length} people in the list:
                </h1>
                <ul>
                    {this.props.people.map(function(person, i){
                        return <Person key={i} person={person} />;
                    })}
                </ul>
            </div>
        );

    }
}

var peopleList1 = [
    {name: "Troy", surname: "Barnes"},
    {name: "Abed", surname: "Nadir"}
];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<PersonList people={peopleList1} />);
```



2 people in the list:

- T. Barnes
- A. Nadir

Next time

In this class we introduced React by creating static content

In the next class we’ll look at adding interactivity to our components

References

Banks, A. & Porcello, E. 2017. *Learning React: Functional Web Development with React and Redux*. O’Reilly Media, Inc.

https://react.dev/reference/react