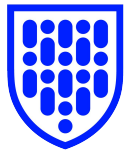
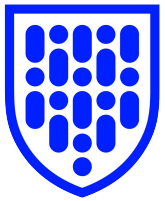
Institute of Data



Software Engineering

Module 6

React JS 1/2



Agenda

Section 1 : React Basics

- Introduction
- JSX
- Props
- Components
- State
- Component Lifecycle
- Handling Events
- Conditional Rendering
- List and Keys
- Form
- Thinking in React

Section 2 : Code Splitting

Introduction

React JS has been the world most popular frontend framework for years. It has been used by many of big companies such as: Facebook, Instagram, Netflix.

There are so many features and benefits to use React in frontend development. It makes web application development easier for developers and also easier for people to learn. Moreover, it is also known to be declarative, powerful, extensible and component-based (almost everything you render on a web page is a component).

Some other features making many developers choose React are that they support mobile web browser and also server-side rendering.

React Introduction

React.js is a JavaScript library that was developed by **Facebook** engineers. It allows one to compose larger UIs into small and isolated pieces of code called **“components”**.

Why React:

- **Speed**
- **Modularity**
- **Scalability**
- **Flexibility**
- **Popularity**

React Basic

What is JSX?

Take a look at the code on the right.

What kind of code do you think it is? Is it HTML? JavaScript?

It is JavaScript.
=> The part that looks like HTML, <h1>Hello world</h1> , is something called ***JSX***.

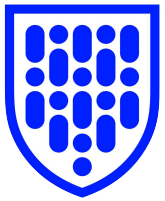
```
const h1 = <h1>Hello world</h1>;
```

Attributes In JSX

JSX elements can have *attributes*, just like HTML elements can.

A JSX attribute is written using HTML-like syntax: a name, followed by an equals sign, followed by a value. The value should be wrapped in quotes, like this:

```
// This <p> tag has an attribute id
of large
const p1 = <p id='large'>foo</p>;
```



React Basic

JSX

It is a **syntax extension** to JavaScript. React uses it to **describe** what the **UI** should look like. JSX puts **markup in the JavaScript**, that makes it look like a **template** language, and also it comes with the full **power of JavaScript**.

```
const element = <h1>Hello, world!</h1>;
```

React Basic

JSX Features

- **Embedding Expressions** in JSX.
  - It **wraps variables** in **curly braces**. (Example Link)

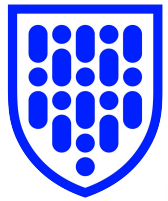
```
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

React Basic

JSX Features

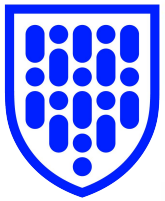
- **Embedding Expressions** in JSX.
  - It **wraps function execution** in **curly braces**. (Example Link)

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}
const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};
const element = (
  <h1>
    Hello, {formatName(user)}!
  </h1>
);
ReactDOM.render(
  element,
  document.getElementById('root')
);
```



- JSX is an **Expression**

After compilation, JSX expressions become **regular JavaScript function calls** and **evaluate to JavaScript objects**. which means you can use JSX inside of `if` **statements** and `for` **loops**, **assign it** to variables, accept it as **arguments**, and **return it** from **functions**.



React Basic

JSX Features

- Specifying **Attributes** with JSX
  - You may use **quotes** to specify **string literals** as attributes:

```
const element = <div tabIndex="0"></div>;
```

- You may also use **curly braces** to embed a **JavaScript expression** in an attribute:

```
const element = <img src={user.avatarUrl}></img>;
```

React Basic

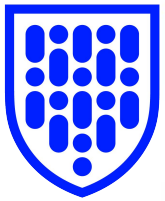
JSX Features

- Specifying **Children** with JSX
  - If a tag is **empty**, you may close it immediately with `/>`, like **XML**

```
const element = <img src={user.avatarUrl} />;
```

- JSX tags may contain **children**:

```
const element = (
<div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
</div>
);
```



React Basic

JSX Features

- JSX **Prevents Injection Attacks**
  By default, React DOM **escapes any values** embedded in JSX before rendering them. Thus it ensures that you can **never inject anything** that is not explicitly written in your application. Everything is converted to a **string** before being **rendered**. This helps prevent **XSS (cross-site-scripting)** attacks.

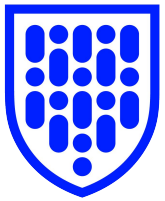
```
const title = response.potentiallyMaliciousInput;
// This is safe:
const element = <h1>{title}</h1>;
```

React Basic

JSX Features

- JSX Represents **Objects**, **Babel** compiles JSX down to `React.createElement()` **calls**.

```
// JSX
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
//Babel transpiles it to
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
// finally it creates an object
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
};
```



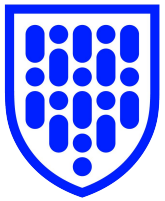
React Basic

What is component?

A component is a small, reusable chunk of code that is responsible for one job. That job is often to render some HTML.

Take a look at the code on the right. This code will create and render a new React component:

```
import React from 'react';
import ReactDOM from 'react-dom';

class MyComponent extends
React.Component {
  render() {
    return <h1>Hello world</h1>;
  }
};

ReactDOM.render(
  <MyComponent />,
  document.getElementById('root')
);
```

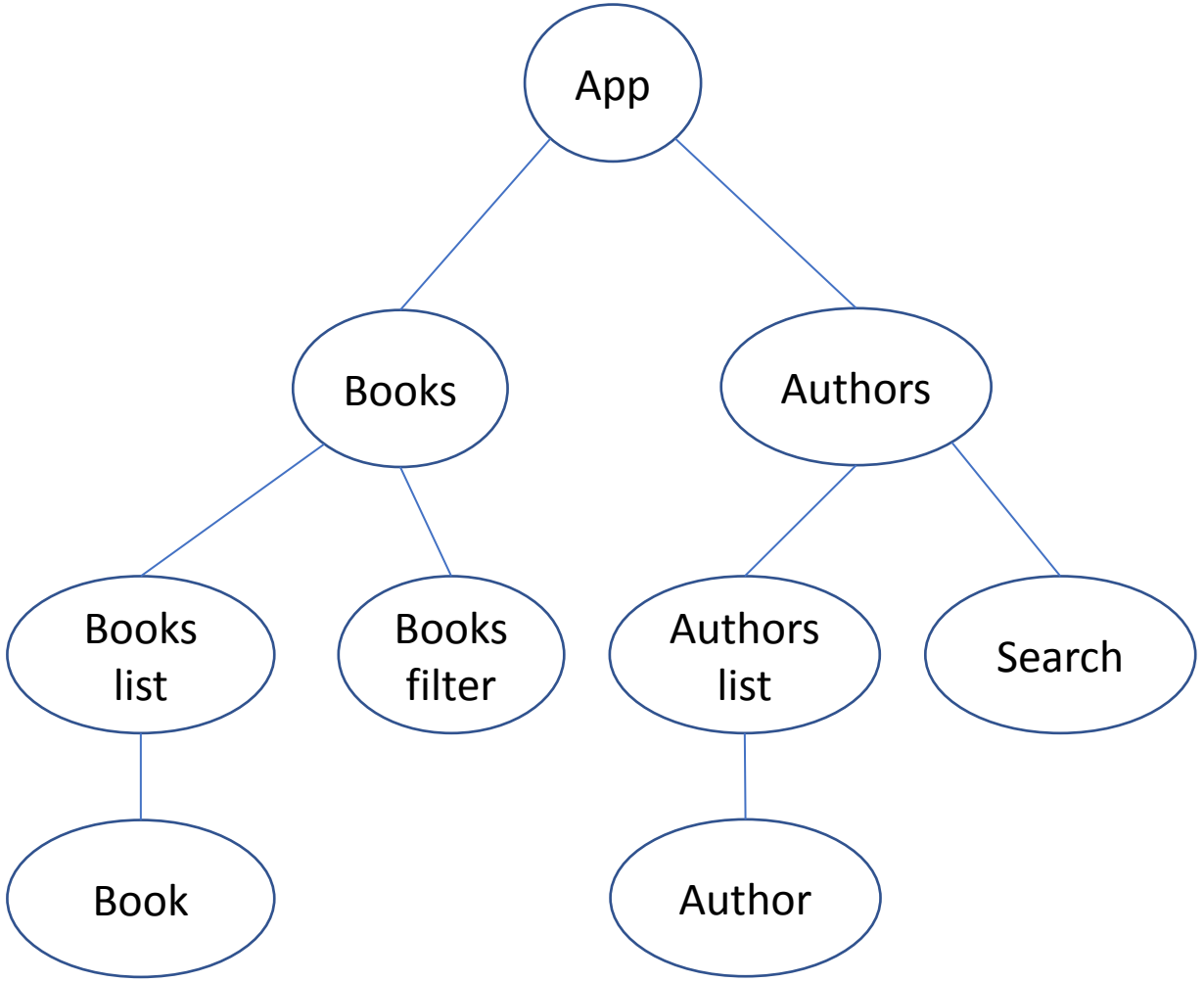


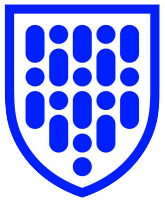
React Basic

A React application can have dozens, even hundreds, of components.

Each of them can only be one very small thing. But when combined, they can make huge and complex application.

Book Shop Application





React Basic

Components and Props

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation

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

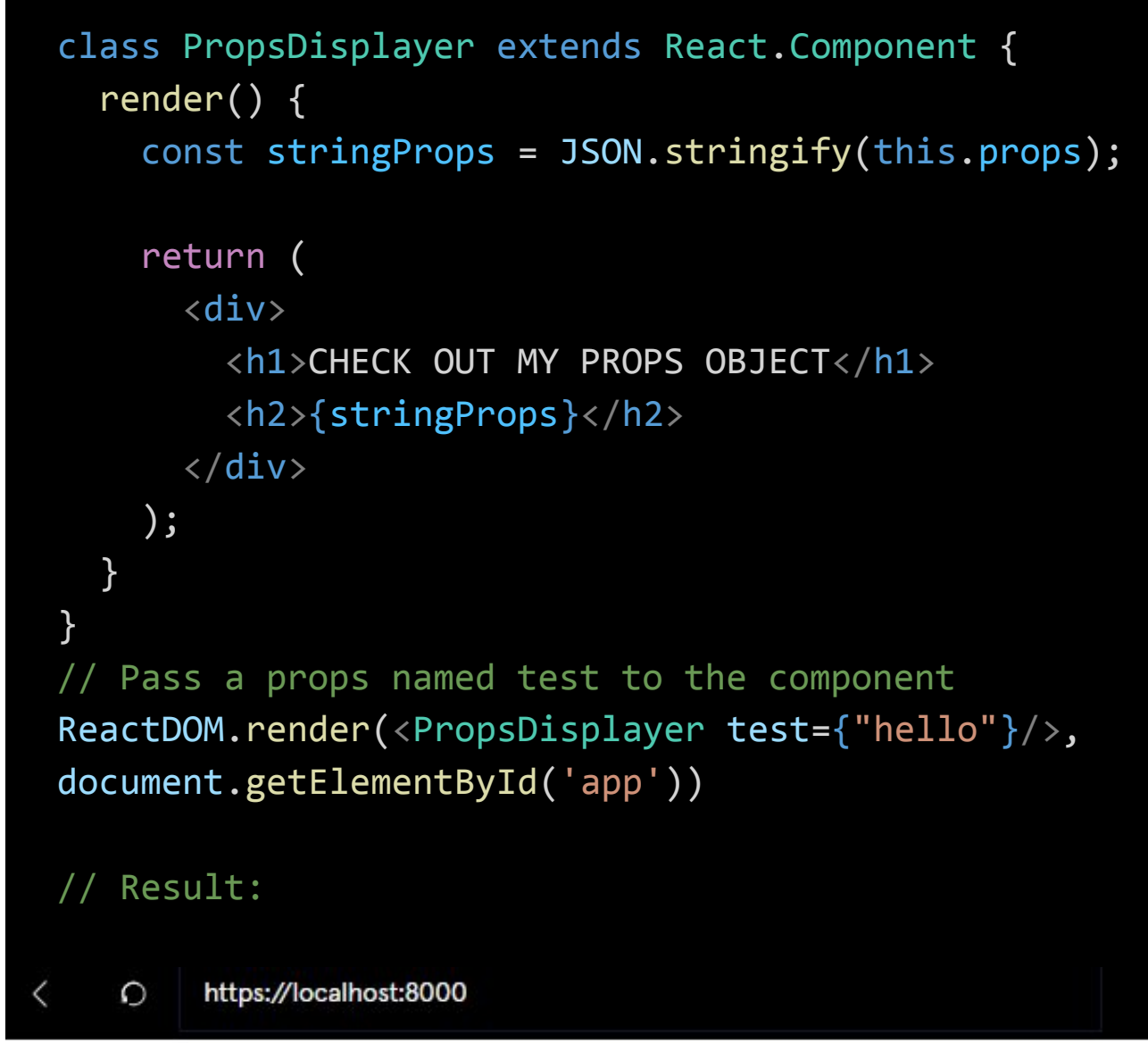
Props

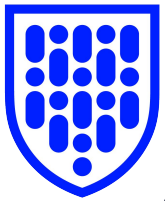
Every component has something called props.

A component's props is an object. It holds information about that component.

To see a component's props object, you use the expression `this.props`. Here's an example of `this.props` being used inside of a render method:

```
class PropsDisplayer extends React.Component {
  render() {
    const stringProps = JSON.stringify(this.props);

    return (
      <div>
        <h1>CHECK OUT MY PROPS OBJECT</h1>
        <h2>{stringProps}</h2>
      </div>
    );
  }
}
// Pass a props named test to the component
ReactDOM.render(<PropsDisplayer test={"hello"}/>,
document.getElementById('app'))

// Result:
```



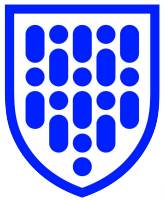


Props

Almost any kind of data can be passed to a component

You can pass `string`, `number`, `array`, `object`, `function`, or even another `component`, to one component.

```
// Pass a string
<MyComponent foo="bar" />

// Pass an array
<Greeting myInfo={["top", "secret", "lol"]}
/>

// Pass multiple props
<Greeting name= "Jonh" town="Flundon" age={2}
haunted={false} />
```



React Basic

Props are Ready-only

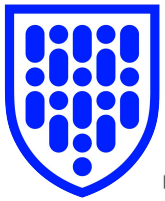
Whether you declare a component as **a function** or **a class**, it must **never modify/mutate** its own **props**.

- **Good** one

```
// pure function
function sum(a, b) {
  return a + b;
}
```

- **Bad** one

```
//impure function, account is mutated / modified
function withdraw(account, amount) {
  account.total -= amount;
}
```



React Basic

The Data Flows Down

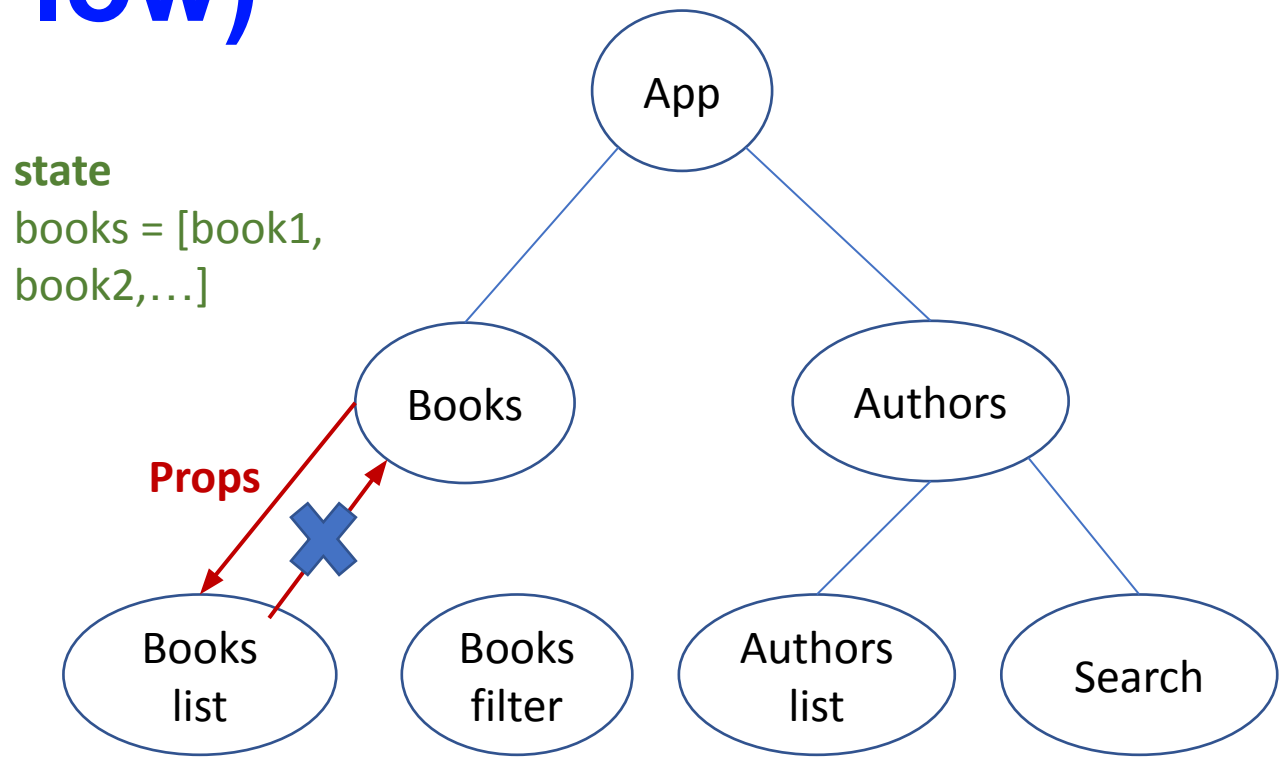
Neither parent nor child components can know if a certain component is **stateful** or **stateless**, and they should **not care** whether it is defined as **a function or a class**.

Any **state** is always **owned by** some **specific component**, and **any data** or **UI** derived from that state can only affect **components "below"** them in the **tree**.

React Basic (Data Flow)

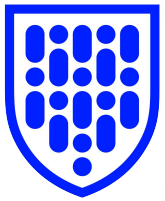
Bookshop App

React organizes components, which has two ingredients: **state** and **props**, into a **hierarchical tree**.



Therefore, a one-way data flow and single state is ensured.

The state *books* are kept track on the **Books** component. It can be passed as **props** to **Books list**, but nothing can be passed in the opposite direction.

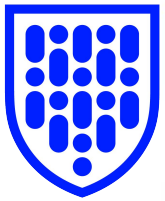


React Basic

Function and Class Components

- **Function components** are literally **JavaScript functions**. they present **valid React components**, because they **accept** a single `props` (which stands for properties) **object argument** with data and **return React elements**.

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

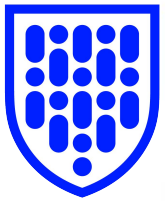


React Basic

Function and Class Components

- **Class components** are **traditionally** the way to create **React Components** by using **ES6 class**

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



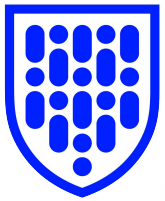
React Basic

Render method

- Syntax

```
ReactDOM.render(element, container[, callback])
```

- Render a React element into the DOM in the **supplied container** and return a **reference to the component**.
- If the React element was **previously rendered** into container, this will perform an **update** on it and **only mutate** the DOM as **necessary** to reflect the **latest** React element.
- If the **optional callback** is provided, it will be **executed** after the component is **rendered** or **updated**.



React Basic

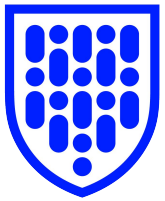
Rendering a component

- **Native** Dom component

```
const element = <div />;
```

- **Use-defined** component, React passes **JSX attributes** and **children** to this component as a **single object**(named `props`).

```
const element = <Welcome name="Sara" />; // use-defined component
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
ReactDOM.render(
  element,
  document.getElementById('root')
);
```



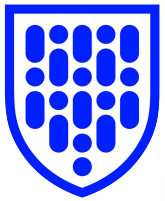
React Basic

The Virtual DOM

One special thing about ReactDOM.render() is that it only updates DOM elements that have changed.

That means that if you render the exact same thing twice in a row, the second render will do nothing.

```
const hello = <h1>Hello world</h1>;

// This will add "Hello world" to the screen:

ReactDOM.render(hello,
document.getElementById('app'));

// This won't do anything at all:

ReactDOM.render(hello,
document.getElementById('app'));
```



React Basic

Composing Components

Components can **refer** to **other components** in their **output**.

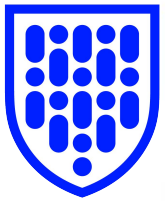
```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}
ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

React Basic

Extracting Components

It means to **split component** into **small components**.

```
// a big Comment component
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}
        />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```



React Basic

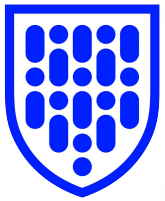
Extracting Components

- Extracting `Avatar`

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}
    />
  );
}
```

- Extracting `UserInfo`

```
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">
        {props.user.name}
      </div>
    </div>
  );
}
```

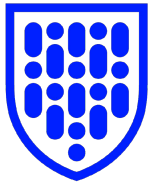


React Basic

Extracting components

The final `Comment` component after extraction. CodePen Example

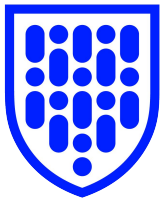
```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```



Institute of Data

Exercise

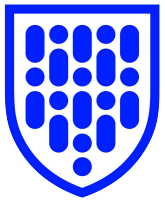
- Create a Greeting component (as a class) which renders a text “Hello World”.
- Pass a prop called ‘name’ down to the Greeting component and render the text ‘Hello <yourname>’ (e.g. Hello John).



React Basic

State

React components will often need ***dynamic information*** in order to render. For example, imagine a component that displays the score of a football game. The score of the game might change over time, meaning that the score is ***dynamic***. Our component needs to know the changes of the score in order to render in a useful way.



React Basic

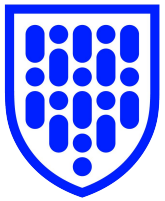
State

A React component can access dynamic information in two ways: **props** and **state**.

Unlike props, a component's state is defined inside itself.

To make a component have state, give the component a state property. This property should be declared inside of a constructor method, like this:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = { mood: 'decent' };
  }

  render() {
    return <div></div>;
  }
}

<Example />
```

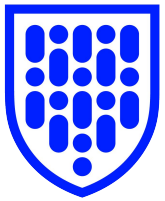


React Basic

State

Let’s have a closer look at what is going on here:

```
class Example extends React.Component {
    constructor(props) {

      // React components always have to
      // call super in their constructors
      // to be set up properly
      super(props);

      // this.state should be an object
      // representing the initial “state”
      // of any component instance
      this.state = {
        mood:   'great',
        hungry: false
      };
    }

    render() {
      return <div></div>;
    }
  }

  <Example />
```



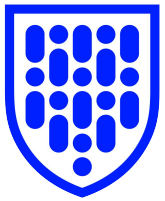
React Basic

Update state

A component changes its state by calling the function `this.setState()`

`this.setState()` takes two arguments: an object that will update the component's state, and a callback called when the update is done.

```
// this.setState() takes an object,
// and merges that object with
// the component's current state
this.setState({ hungry: true });
```



React Basic

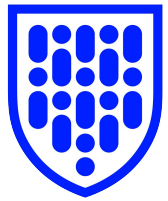
Update state

When we update the `hungry` property in the state, the `mood` part is not affected.

`this.setState()` *merges* the new provided object with the one in the component's current state.

If the new object *does not* have any properties matching with those in the current state, those properties remain unchanged.

```
// Initial state:
{
  mood:   'great',
  hungry: false
}

// Update state:
this.setState({ hungry: true });

// Result:
{
  mood:   'great',
  hungry: true
}
```



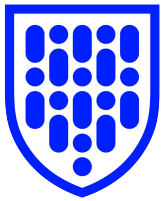
React Basic

Update state using a function

The most common way to call `this.setState()` is to call a custom function containing it.

For example:

```
class Weather extends React.Component {
  constructor(props) {
    super(props);
    this.state = { weather: 'sunny' };
    this.windBlows = this. windBlows.bind(this);
  }

  windBlows() {
    this.setState({ weather: 'windy' });
  }
}
```



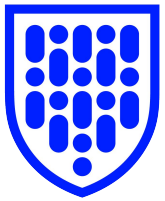
React Basic

Update state using a function

Why do we need to do this:

Because:
In React, whenever you define an event handler that uses `this`, you need to add `this.methodName = this.methodName.bind(this)` to your constructor function.

```
this.windBlows = this.windBlows.bind(this);
```



React Basic

Update state & Re-render

Any time that you call `this.setState()`, `.render()` is AUTOMATICALLY CALLED as soon as the state has changed.

=> You cannot directly call `this.setState()` inside `.render()` which will cause an infinite loop of re-rendering.

```
// Initial state:
{
  mood:    'great',
  hungry: false
}

// Update state:
this.setState({ hungry: true });

// Result:
{
  mood:    'great',
  hungry: true
}
```

React Basic

Lifting State Up

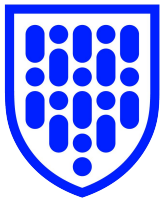
In React, **sharing state** is accomplished by **moving** it up to the **closest common ancestor of the components** that need it. This is called **"lifting state up"**.

```
const TemperatureInput = (props) =>  {
  const handleChange = (e) => {
    props.onTemperatureChange(e.target.value); // the state of input value is lifted up by the callback function from props
  }

  const temperature = props.temperature;
  const scale = props.scale;
  return (
    <fieldset>
      <legend>Enter temperature in {scaleNames[scale]}:</legend>
      <input value={temperature}
             onChange={handleChange} />
    </fieldset>
  );
}
```

Institute of Data

Exercise

- Create a button for your Greeting class component which replaces “Hello World” with “Hello <your name>” when clicked.



Component Lifecycle

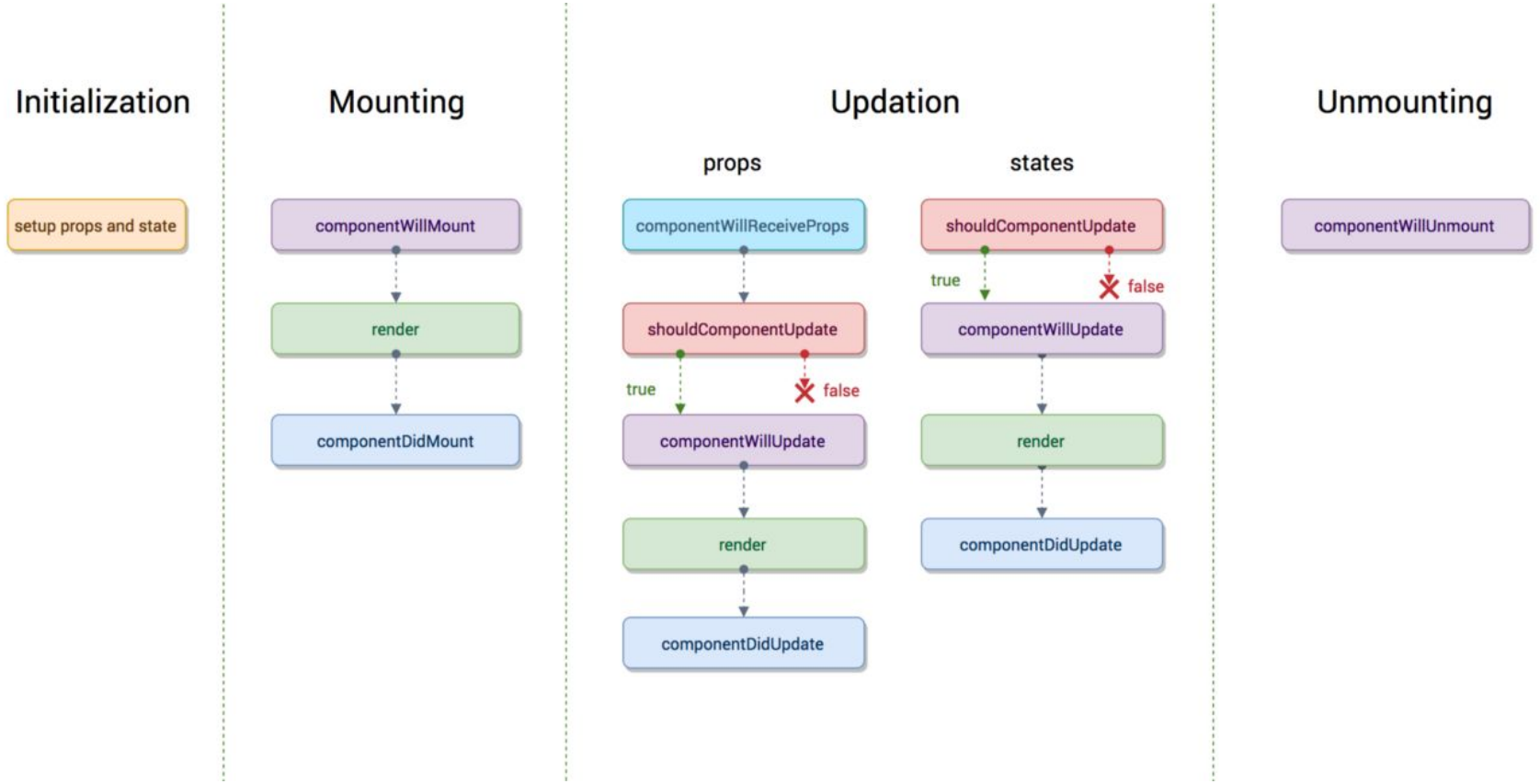
1. ***Mounting***, loading the component for the first time

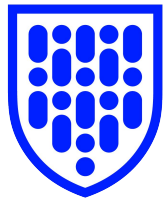
2. ***Updating***, resulting from changed state or changed props

3. ***Unmounting***, removing the component from the DOM

React Basic

Component Lifecycle





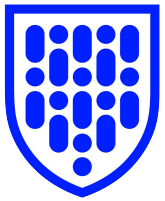
React Basic

Mounting methods

constructor()

This lifecycle method is the first one that is called and is usually where you will set the state of your component.

```
import React from 'react';

class Person extends React.Component {
  constructor() {
    super();
    this.state = {
      name: 'John Doe',
      age: 14,
    }
  }
}
```



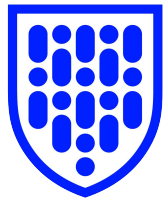
React Basic

Mounting methods

render()

This method is called after the constructor method has finished. It is the only lifecycle method required by React. This method is in charge of displaying HTML markup to the user.

```
import React from 'react’;

class App extends React.Component
{
  render() {
    return (
      <div>
        <h1>Hello</h1>
      </div>
    )
  }
}
```



React Basic

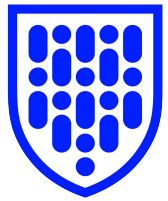
Mounting methods

componentDidMount()

This method is available after the component has mounted.

That is after the HTML from the *render()* has finished loading. It is called once in the component's lifecycle, and it signals that the component and all of its sub-component have rendered properly.

```
import React from 'react';

class App extends React.Component {
  componentDidMount() {
    fetch('http://example.com')
      .then(response => response.json())
      .then(json => console.log(json));
  }
}
```



React Basic

Unmounting methods

componentWillUnmount()

This method is called right before a component is removed from the DOM.

This method is useful to perform cleanups that should be done such as canceling network requests and removing event listeners.

```
import React from 'react';

class App extends React.Component
{
  componentDidMount() {
    this.timer = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timer);
  }
}
```

React Basic

State and LifeCycle

Example of Clock

To render a `Clock` **without** using **state** and **lifeCycle**

```
function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

React Basic

State and LifeCycle

To render a `Clock` **with state** and **lifeCycle** in **Class** component

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }
  componentWillUnmount() {
    clearInterval(this.timerID);
  }
  tick() {
    this.setState({
      date: new Date()
    });
  }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

React Basic

Handling Events

- **React events** are named using **camelCase**, rather than **lowercase**. With JSX you pass **a function** as the **event handler**, rather than **a string**.

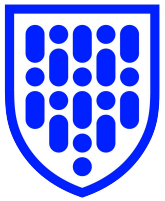
```
// in native DOM
<button onclick="activateLasers()">
  Activate Lasers
</button>

// in React
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

React Basic

Handling Events

- you **cannot** return **false** to **prevent default behavior** in React. You must call **preventDefault** explicitly.

```
// native Dom
<form onsubmit="console.log('You clicked submit.'); return false">
  <button type="submit">Submit</button>
</form>

// in React
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('You clicked submit.');
  }

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
}
```



React Basic

Conditional Rendering

Use JavaScript operators like `if` or the **conditional operator** to create elements representing the current state.

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

ReactDOM.render(
  // Try changing to isLoggedIn={true}:
  <Greeting isLoggedIn={false} />,
  document.getElementById('root')
);
```

React Basic

Conditional Rendering

You can use **variables to store elements**. This can help you **conditionally render** a part of the component while the **rest** of the output **doesn't change**.

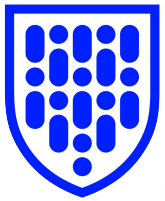
```
class LoginControl extends React.Component {
  render() {
    const isLoggedIn = this.props.isLoggedIn;
    let button;
    if (isLoggedIn) {
      button = <LogoutButton  />;
    } else {
      button = <LoginButton />;
    }
    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
}
```

React Basic

Conditional Rendering

use **inline logic** operator `&&` to **"shortcircuit"** `if` condition.

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}

const messages = ['React', 'Re: React', 'Re:Re: React'];
ReactDOM.render(
  <Mailbox unreadMessages={messages} />,
  document.getElementById('root')
);
```

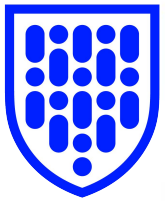


React Basic

Conditional Rendering

use **tertiary** operator `condition ? true : false` to mimic `if /else` statement

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn
        ? <LogoutButton onClick={this.handleLogoutClick} />
        : <LoginButton onClick={this.handleLoginClick} />
      }
    </div>
  );
}
```



React Basic

Conditional Rendering

Return `null` to **prevent component** from rendering.

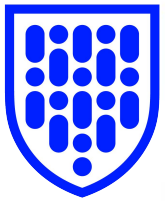
```
function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }

  return (
    <div className="warning">
      Warning!
    </div>
  );
}
```

React Basic

Lists and Keys

- Basic List Components

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```



React Basic

Lists and Keys

- `Keys` in the list

The **key** should be a **string** that **uniquely identifies** a list item among its **siblings**. Most often you would use **IDs** from your data **as key**. Keys should be **stable, predictable, and unique**.

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>
    {todo.text}
  </li>
);
```

React Basic

Lists and Keys

- **Extracting** components with Keys

The **key** makes sense only **inside the loop**. A good **rule of thumb** is that elements inside the `map()` call need **keys**.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Correct! Key should be specified inside the array.
    <ListItem key={number.toString()} value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

React Basic

Forms

Controlled Components

An **input form element** whose **value** is **controlled by React** in this way is called a **controlled component**. The input's value is always **driven** by the **React state**.

- Handling **Single** Input

```
const NameForm = () => {
  const [value, setValue] = React.useState('');
  const handleChange = (event) =>  {
    setValue(event.target.value);
  }
  return (
    <form>
      <label> Name:<input type="text" value={value} onChange={handleChange} /></label>
    </form>
  );
}
```

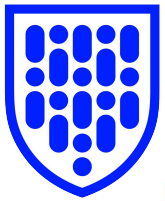
React Basic

Forms

Controlled Components

- Handling **Multiple** Inputs
  We add a **name attribute** to each element and **get the value** by using `event.target.name`.

```
const Reservation = () => {
  const [formState, setFormState] = React.useState({ isGoing: true, numberOfGuests: 2 });
  const handleInputChange = (event) => {
    const target = event.target;
    const value = target.type === "checkbox" ? target.checked : target.value;
    setFormState((pre) => ({ ...pre, [target.name]: value }));
  };
  return (
    <form>
      <label>
        Is going:<input name="isGoing" type="checkbox" checked={formState.isGoing} onChange={handleInputChange} />
      </label>
      <label>
        Number of guests: <input name="numberOfGuests" type="number" value={formState.numberOfGuests} onChange={handleInputChange} />
      </label>
    </form>
  );
};
```



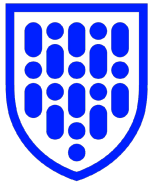
React Basic

Composition vs Inheritance

React has a powerful **composition model**, and we recommend using **composition instead of inheritance** to reuse code between components.

Such components use the special **children prop** to pass **children elements** directly into **their output**.

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
```



Institute of Data

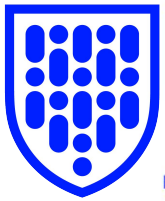
Exercise

- Create a component which renders emojis.
- Provide a button in the component to switch between 2 emojis.

React Basic

Thinking in React

- **Step 1**: **Break** The **UI** Into A Component **Hierarchy**



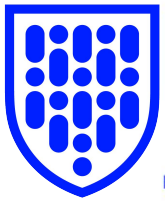
React Basic

Thinking in React

1. **FilterableProductTable (orange)**: contains the entirety of the example
2. **SearchBar (blue)**: receives all user input
3. **ProductTable (green)**: displays and filters the data collection based on user input
4. **ProductCategoryRow (turquoise)**: displays a heading for each category
5. **ProductRow (red)**: displays a row for each product

The Hierarchy

- **FilterableProductTable**
  - **SearchBar**
    - **ProductTable**
      - **ProductCategoryRow**
      - **ProductRow**



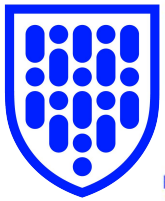
React Basic

Thinking in React

- **Step 2**: Build A **Static Version** in React.

To build a **static version** of your app that renders your data model, you will want to build components that reuse other components and **pass data using props**. props are a way of **passing data from parent to child**.

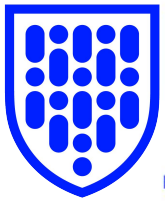
Don't use state at all to build this static version. State is **reserved only for interactivity**, that is, data that changes over time.



React Basic

Thinking in React

- **Step 3**: Identify The **Minimal (but complete) Representation** Of UI **State**.
  - Is it passed in from a parent **via props**? If so, it probably is **not** state.
  - Does it remain **unchanged over time**? If so, it probably is **not** state.
  - Can you **compute it** based on any **other state or props** in your component? If so, it is **not** state.
  - Does it change **after the user interacts with UI** elements? if so, yes it is the state.



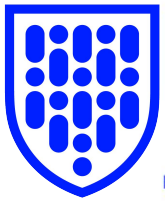
React Basic

Thinking in React

- **Step 4**: Identify **Where** Your State Should **Live**.

We need to **identify** which component **mutates, or owns**, this **state**.

1. Identify every component that **renders** something **based on** that state.
2. Find a **common owner component** (a single component above all the components that need the state in the hierarchy).
3. Either the **common owner** or another **component higher up** in the hierarchy should **own the state**.
4. If you **can't find a component** where it makes sense to own the state, **create a new component** solely for **holding the state** and **add it** somewhere in the **hierarchy** above the common owner component.



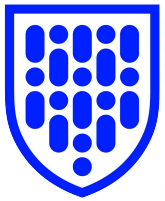
React Basic

Thinking in React

- **Step 5**: Add **Inverse Data Flow**.

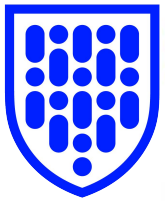
So far, we have built an app that renders correctly as a **function of props** and **state** flowing down the hierarchy. Now it is time to support data **flowing the other way**.

React makes this data **flow(reverse flow)** explicit to help you understand how your program works, but it does require a little more typing **than traditional two-way data binding**.



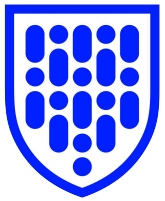
Code Splitting

- **Bundling** is great, but as your **app grows**, your bundle will **grow too**. Especially if you are including **large third-party** libraries. You need to keep an eye on the code you are including in your bundle so that you don't accidentally make it so **large** that your app **takes a long time** to load.
- To **avoid** winding up with a **large bundle**, it is good to get ahead of the problem and start **"splitting"** your bundle. Code-Splitting is a **feature supported** by bundlers like **Webpack, Rollup and Browserify (via factor-bundle)** which can create **multiple bundles** that can be **dynamically loaded** at runtime.



Code Splitting

- **Code-splitting** your app can help you **"lazy-load"** just the things that are **currently needed** by the user, which can **dramatically improve the performance** of your app. While you haven't reduced the overall amount of code in your app, you have **avoided loading code** that the user may **never need**, and **reduced the amount of code** needed during the **initial load**.



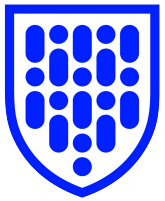
Code Splitting

When you use React.js, every JavaScript file in your application is unknown to each other by default.

Therefore, without telling one component about the other, they cannot relate.

For example:

```
import React from 'react';
import ReactDOM from 'react-dom';


class ProfilePage extends React.Component {
  render() {
    return (
      <div>
   // ProfilePage component has no idea what is
   // <NavBar />
        <NavBar />
        <h1>All About Me!</h1>
        <p>I like movies and blah blah blah b
lah blah</p>
        <img src="https://content.codecademy.
com/courses/React/react_photo-monkeyselfie.jp
g" />
      </div>
    );
  }
}
```



Code Splitting

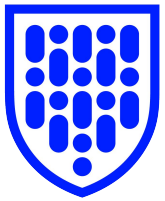
Import

If you want to use a variable that's declared in a different file, such as ***NavBar***, then you have to *import* the variable that you want. To import a variable, you can use an import statement.

If you use an import statement, and the string at the end begins with either a dot or a slash, then import will treat that string as a *filepath*. *import* will follow that filepath and import the file that it finds.

You will see people often use two slightly different syntaxes to import:

```
import { NavBar } from './NavBar.js';

// If your filepath doesn't have a file extension, then //
".js" is assumed. So, the above example could be
// shortened
import { NavBar } from './NavBar';



import React from 'react';
```



Code Splitting

Export

Alright! You've learned how to use import to grab a variable from a file other than the file that is currently running.

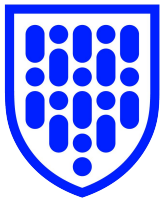
When you ***import*** a variable from a file not exposing itself is not enough. Therefore, you need ***export*** to make it importable by other files.

export comes from ES6's module system, just like ***import*** does. ***export*** and ***import*** are meant to be used together, and you rarely see one without the other.

There are 2 different types of ***export***: **`named`** and **`default`**. You are allowed multiple named exports per module but only one default export.

In one file, include the keyword export immediately before something that you want to export. That something can be any top-level `var`, `let`, `const`, `function`, or `class`:

```
// Named exports
export const name = "John Doe";
export { name };
export { name as fullName };

// Default exports
export default name;
export { name as default };
```



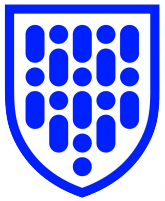
Code Splitting

Import & Export

Named exports are useful to export several values. They must be imported within curly braces with the same name of the corresponding object.

A default export can be imported with any name, for example:

```
// named export from file test.js
let k;
export k = 42;

// another file
import { k } from './test’;
console.log(k);         // this will log 42

// ------------------------------------ //

// default export from the file test.js
let k;
export default k = 42;

// another file
import m from './test’;
// we can use import m instead of import k,
// because k was default exported.

console.log(m);         // this will log 42
```



Code Splitting

Dynamic import

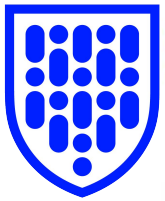
The best way to introduce **code-splitting** into your app is through the **dynamic** `import()` syntax.

- **Static** Import

```
import { add } from './math';

console.log(add(16, 26));
```

- **Dynamic** Import

```
import("./math").then(math => {
  console.log(math.add(16, 26));
});
```



Code Splitting

React.lazy

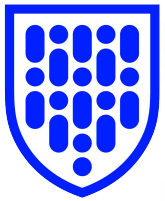
The `React.lazy` function lets you render a **dynamic import** as a **regular component**.

- **Static** Import Component

```
import OtherComponent from './OtherComponent';
```

- **Dynamic** Import Component with `lazy`

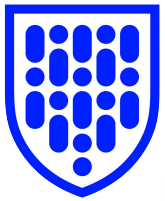
```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```



Code Splitting

- `lazy` component should be **rendered inside** a `Suspense` component.

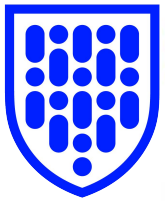
```
import React, { Suspense } from 'react';
const OtherComponent = React.lazy(() => import('./OtherComponent'));
function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```



Code Splitting

- **Error Handling** with `Error Boundaries`.

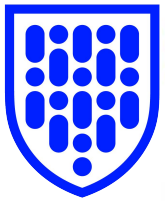
```
import React, { Suspense } from 'react';
import MyErrorBoundary from './MyErrorBoundary';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));
const MyComponent = () => (
  <MyErrorBoundary>
    <Suspense fallback={<div>Loading...</div>}>
      <section>
        <OtherComponent />
        <AnotherComponent />
      </section>
    </Suspense>
  </MyErrorBoundary>
);
```



Code Splitting

- **Route-based** code splitting

```
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Switch>
        <Route exact path="/" component={Home}/>
        <Route path="/about" component={About}/>
      </Switch>
    </Suspense>
  </Router>
);
```



Code Splitting

- **Named** Exports

  `React.lazy` currently **only** supports **default** exports. If the module you want to import uses **named exports**, you can create an **intermediate module** that **reexports it as the default**

```
// ManyComponents.js
export const MyComponent = /* ... */;
export const MyUnusedComponent = /* ... */;
// MyComponent.js
export { MyComponent as default } from "./ManyComponents.js";
// MyApp.js
import React, { lazy } from 'react';
const MyComponent = lazy(() => import("./MyComponent.js"));
```

Institute of Data

Exercise

- Split the emoji into a separate component and import it in the App.
- Create a Calculator component which takes 2 numbers and the operator provided by the user and display the result.
- Import it into App to be rendered under our Emoji component.