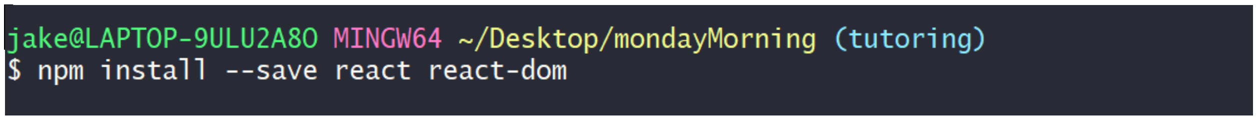
React Basics Walkthrough

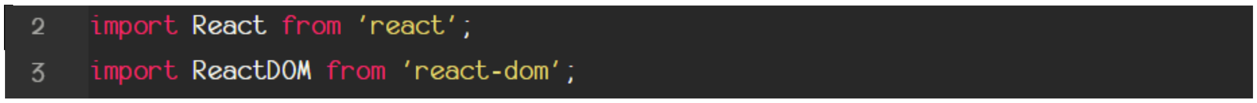
Begin with react:

First install react and react-dom from npm. React gives us access to React components and functionlity (as we will see later), and react-dom, renders these components to the DOM.



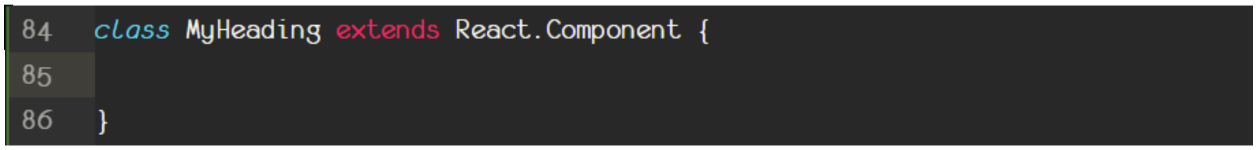
```
jake@LAPTOP-9ULU2A8O MINGW64 ~/Desktop/mondayMorning (tutoring)
$ npm install --save react react-dom
```

Import these modules on your project.



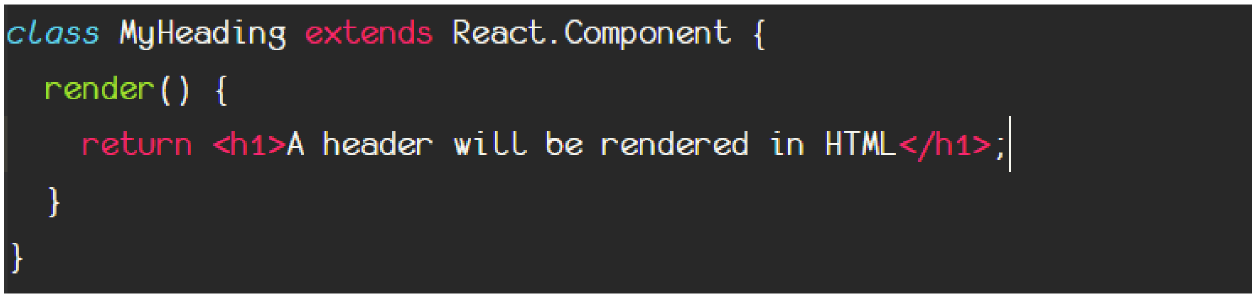
```
import React from ‘react’;
import ReactDOM from ‘react-dom’;
```

Lets create our first component and try to render some HTML to it (actually is JSX, a mixed version between JavaScript and HTML)



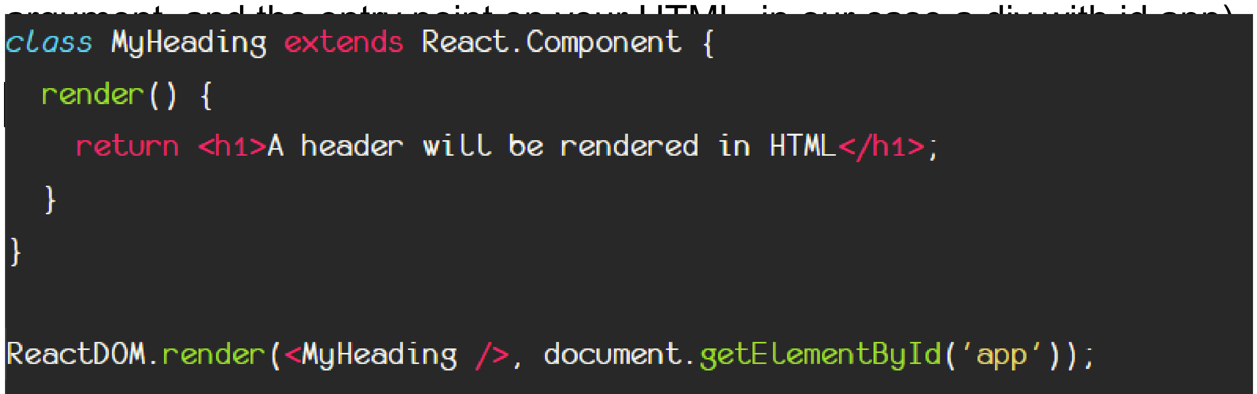
```
class MyHeading extends React.Component {

}
```

Inside the   MyHeading we will try to render some HTML. We need the render function for that. By default the render function should return a JSX structure, that will be compiled to HTML by Webpack.



```
class MyHeading extends React.Component {
  render() {
    return <h1>A header will be rendered in HTML</h1>;
  }
}
```

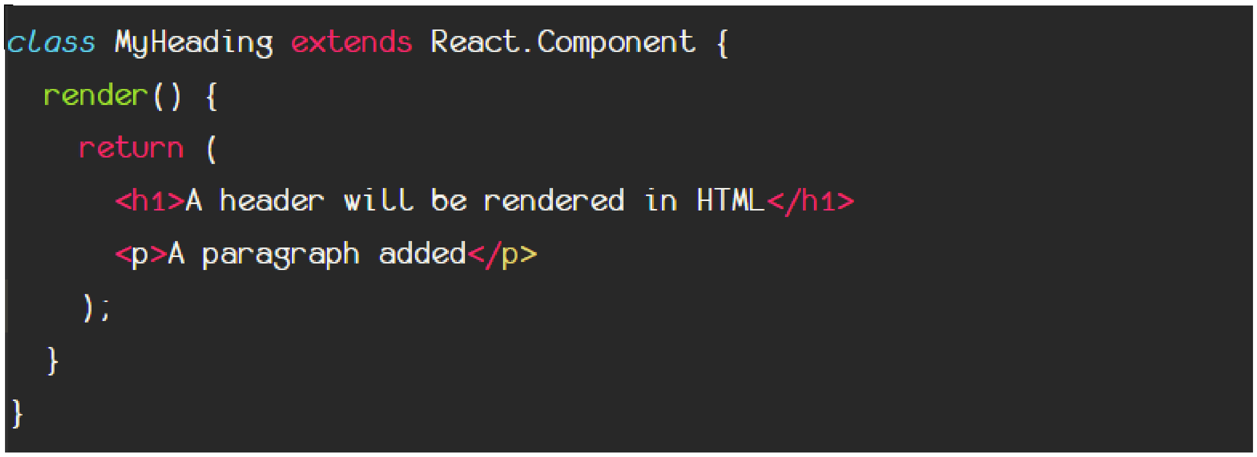
Mount whatever this component renders to the DOM by using the ReactDOM.render method. Specifiy the component (or some JSX as first argument, and the entry point on your HTML, in our case a div with id app)



```
class MyHeading extends React.Component {
  render() {
    return <h1>A header will be rendered in HTML</h1>;
  }
}

ReactDOM.render(<MyHeading />, document.getElementById(‘app’));
```

What if a component renders more than one element, a whole structure for example: We could use a parenthesis in order to specify a multi-line structure return statement.

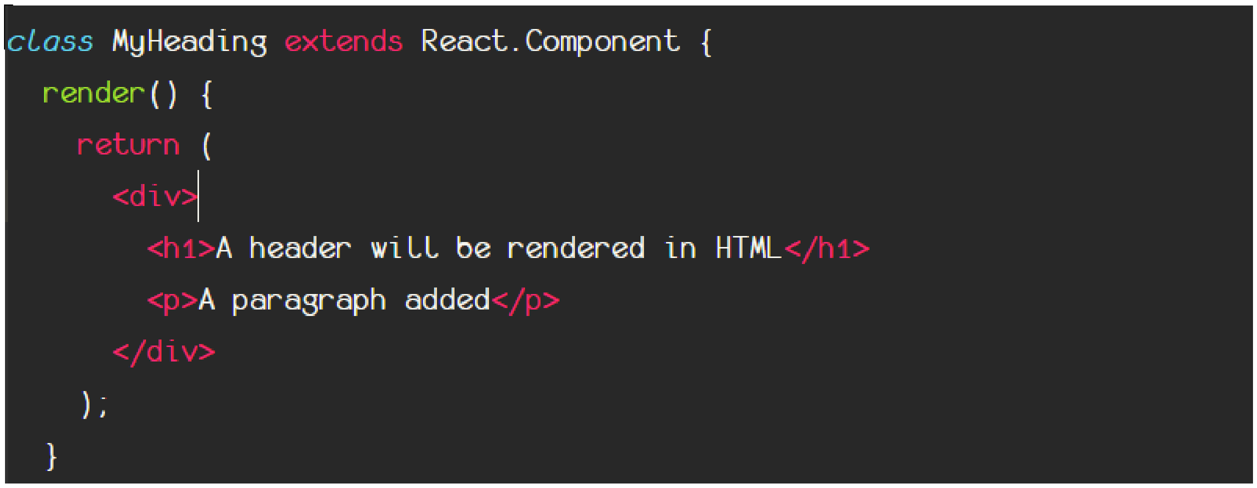
Would this work?



```
class MyHeading extends React.Component {
  render() {
    return (
      <h1>A header will be rendered in HTML</h1>
      <p>A paragraph added</p>
    );
  }
}
```

The answer is unfortunately still no. Every component must return ONE single outermost element, whatever is inside can go as deep as you want. But outside is going to be only one wrapper element.

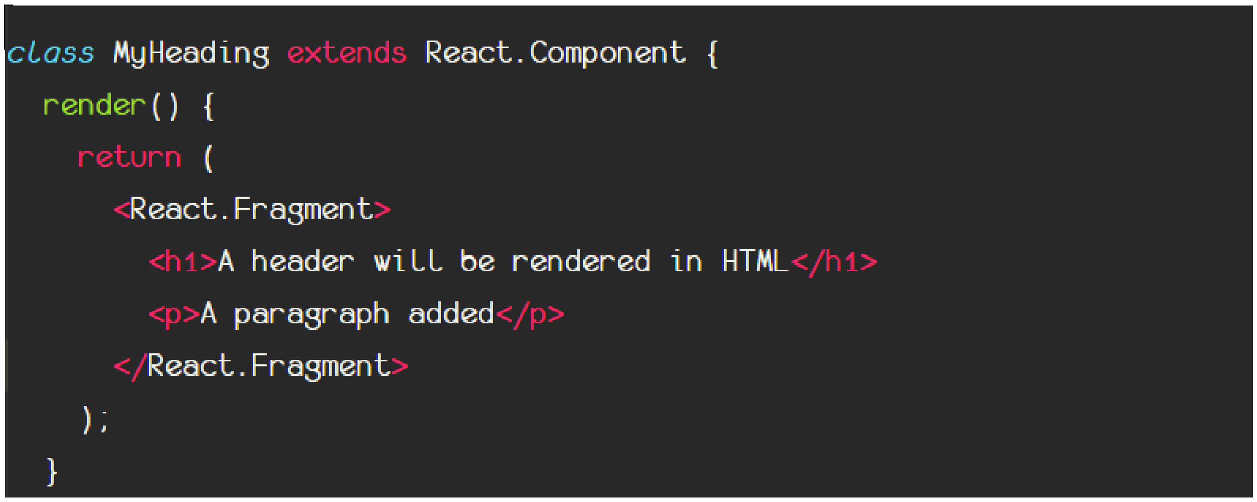
So we could re-write this functionality such:



```
class MyHeading extends React.Component {
  render() {
    return (
      <div>
        <h1>A header will be rendered in HTML</h1>
        <p>A paragraph added</p>
      </div>
    );
  }
```

That would work perfectly, because the MyHeading component basically renders one DIV element with some HTML inside of it. But the outer element is always one.

If you dont want to literally return a div or a specific element that has a meaning to HTML, but just a wrapper, React gives you this opportunity with React.Fragment. That acts like a wrapper but will be ignored during compile from the browser.

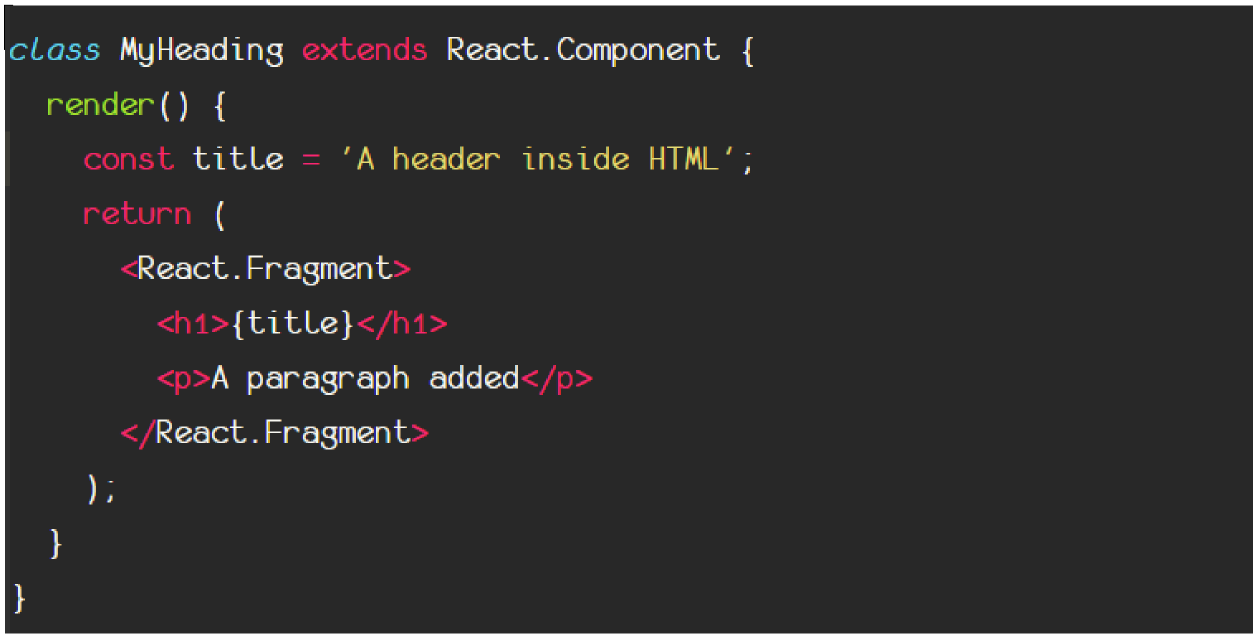


```
class MyHeading extends React.Component {
  render() {
    return (
      <React.Fragment>
        <h1>A header will be rendered in HTML</h1>
        <p>A paragraph added</p>
      </React.Fragment>
    );
  }
```

Interpolate HTML

What if we want to add some JS functionality inside our HTML. Normally you need to write functions, to grab some elements by selectors, and then change the innerHTML or the innerText etc.

Well react provides a much neater way of doing these things and interpolating JS expressions inside JSX. As it seems, in React whatever needs to be evaluated has to be inside curly brace { }.

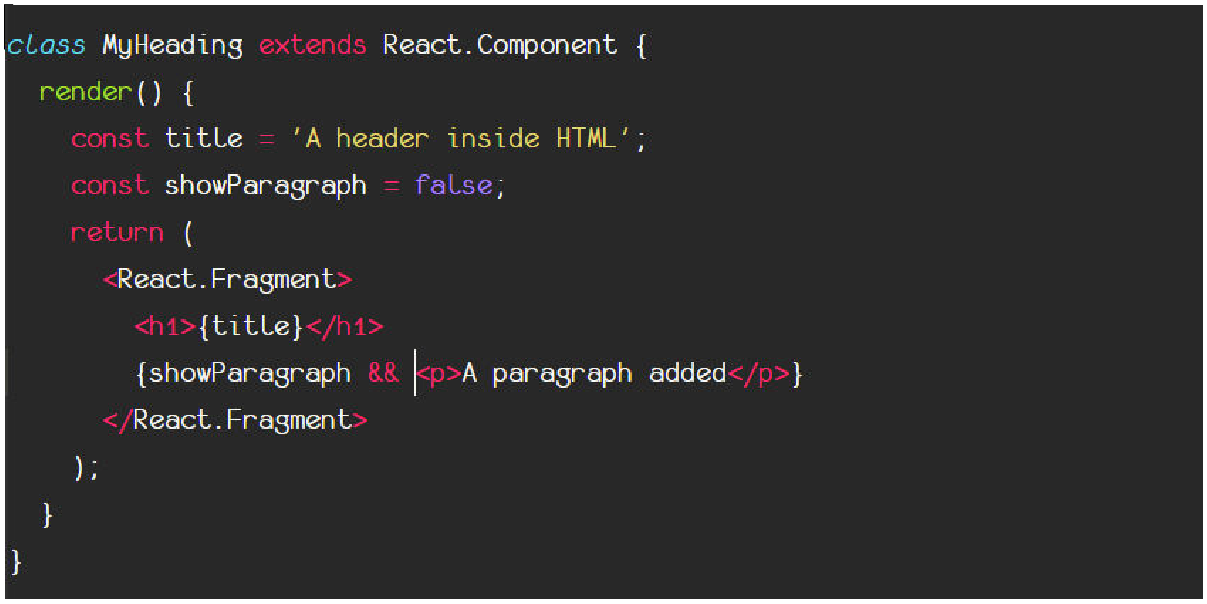


```
class MyHeading extends React.Component {
  render() {
    const title = 'A header inside HTML';
    return (
      <React.Fragment>
        <h1>{title}</h1>
        <p>A paragraph added</p>
      </React.Fragment>
    );
  }
}
```

Conditional rendering

Sometimes we want to evaluate a boolean value, and based upon it to decide if an element (or even a whole component) will be rendered. There are many ways to do that: Ternary operator, if else statements etc. In this case we will use the && operator that React provides in order to make decisions. Syntax has to be inside { } as we saw any time we need to evaluate a JS expression inside JSX.

In the case above, the paragraph will be in the DOM only in case showParagraph evaluates to true (has a truthy value).

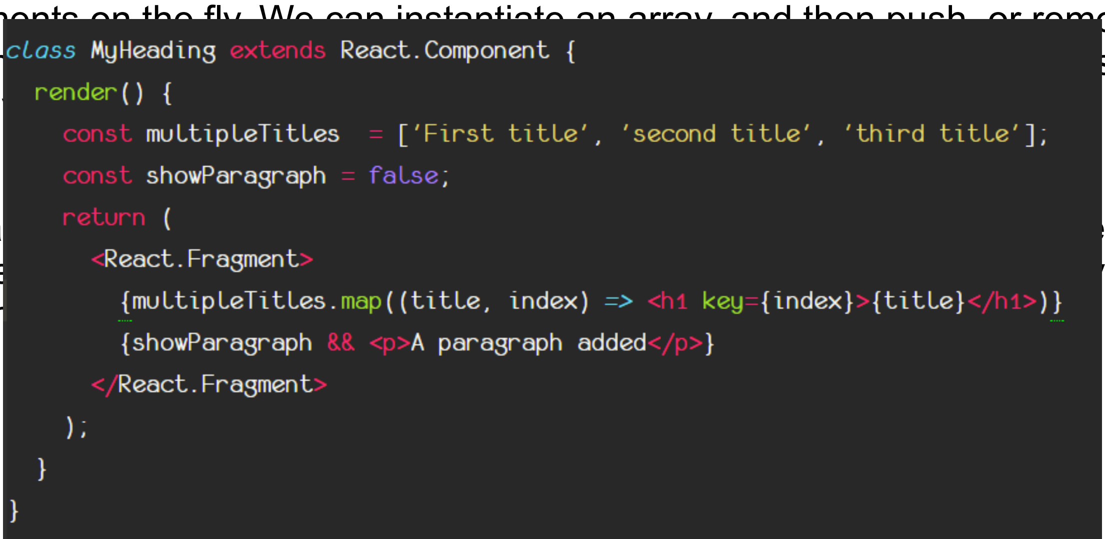


```
class MyHeading extends React.Component {
  render() {
    const title = 'A header inside HTML';
    const showParagraph = false;
    return (
      <React.Fragment>
        <h1>{title}</h1>
        {showParagraph && <p>A paragraph added</p>}
      </React.Fragment>
    );
  }
}
```

Dynamic Rendering

Sometimes we dont know how many elements we would like to render. The react provides a very nice way by using the map function in order to create elements on the fly. We can instantiate an array, and then push, or remove
elem s
man

The
para r
the e e to
do th



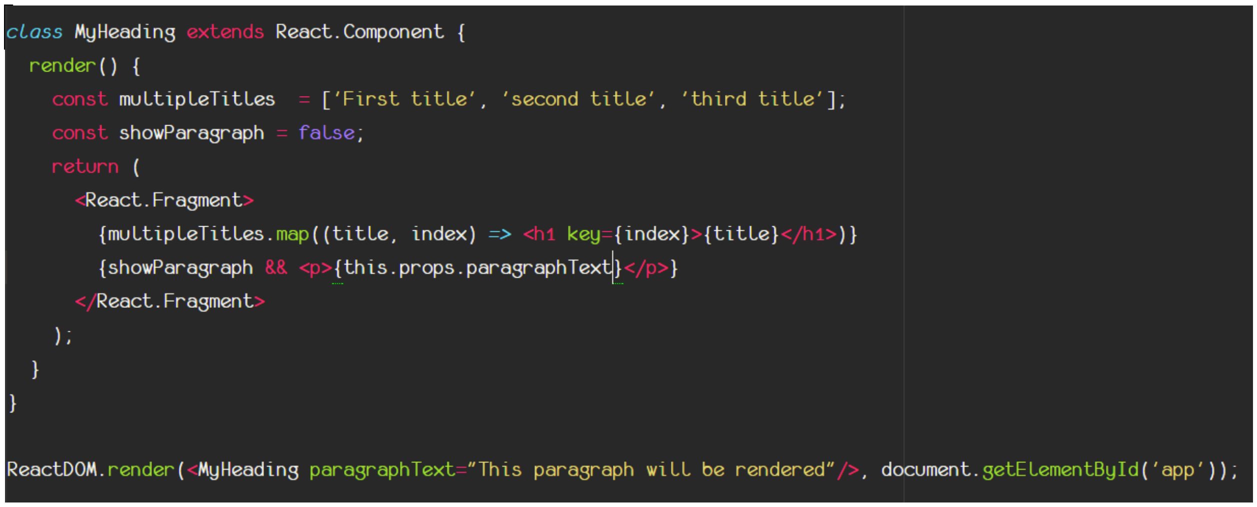
```
class MyHeading extends React.Component {
  render() {
    const multipleTitles  = ['First title', 'second title', 'third title'];
    const showParagraph = false;
    return (
      <React.Fragment>
        {multipleTitles.map((title, index) => <h1 key={index}>{title}</h1>)}
        {showParagraph && <p>A paragraph added</p>}
      </React.Fragment>
    );
  }
}
```

multipleTitles variable. As convention we use the index of every map function.

Components props

Every component is like a template. This template although fixed, can vary, and every component that comes from the same class can differentiate itself thourgh props. Consider that all humans come from the <Human /> component. That means that all humans share some characteristics. A date they were born, a nationality, an age, a name etc. These characteristics are different though from human to human. The template is the same, the content no. These characteristics that are unique from components instance to another components instance are called props of the component.

Components props can be accessed anytime inside the component through the {this.props.nameOfPropHere} property. You can have as many as you want. Lets see an example below:



```
class MyHeading extends React.Component {
  render() {
    const multipleTitles  = ['First title', 'second title', 'third title'];
    const showParagraph = false;
    return (
      <React.Fragment>
        {multipleTitles.map((title, index) => <h1 key={index}>{title}</h1>)}
        {showParagraph && <p>{this.props.paragraphText}</p>}
      </React.Fragment>
    );
  }
}

ReactDOM.render(<MyHeading paragraphText="This paragraph will be rendered"/>, document.getElementById('app'));
```

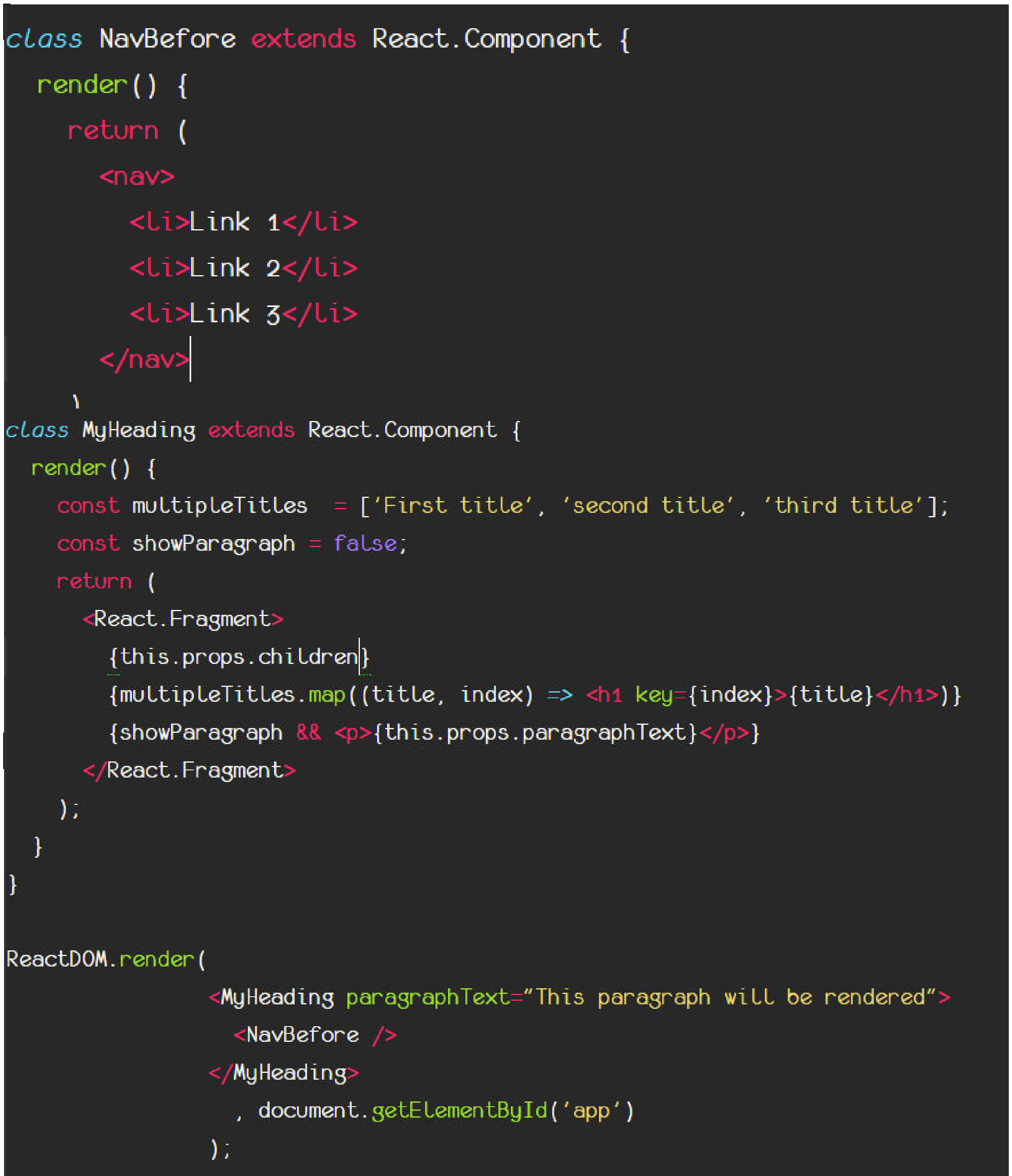
The props are passed always when the component mounts to the DOM (that means create a new object of this class, with these props, in this case the paragraphText).

Then inside the component we use this string in order to put it to the paragraph.
You can create and render many <MyHeading /> components with different props and change the behavior and the appearance of it.

Children Components

A very common use is that components, except JSX can also render other compnents. That means that a component acts as a parent-component and inside renders other components and whatever these render.

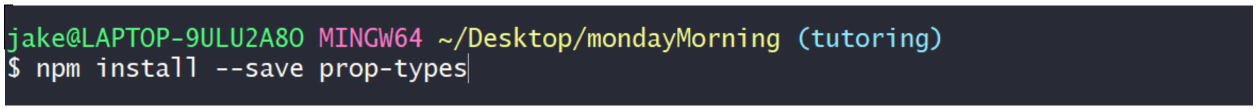
A syntax would look like that:



```
class NavBefore extends React.Component {
  render() {
    return (
      <nav>
        <li>Link 1</li>
        <li>Link 2</li>
        <li>Link 3</li>
      </nav>
    )
class MyHeading extends React.Component {
  render() {
    const multipleTitles  = ['First title', 'second title', 'third title'];
    const showParagraph = false;
    return (
      <React.Fragment>
        {this.props.children}
        {multipleTitles.map((title, index) => <h1 key={index}>{title}</h1>)}
        {showParagraph && <p>{this.props.paragraphText}</p>}
      </React.Fragment>
    );
  }
}

ReactDOM.render(
                <MyHeading paragraphText="This paragraph will be rendered">
                  <NavBefore />
                </MyHeading>
                  , document.getElementById('app')
                );
```

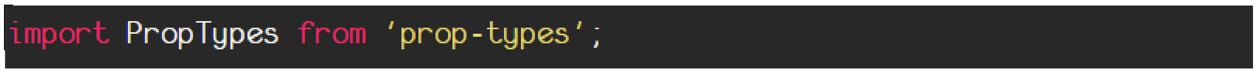
PropTypes are not mandatory section of react but is always nice to have for performance and for prop validation purposes.

First we must install the separate prop-types module from npm registry.



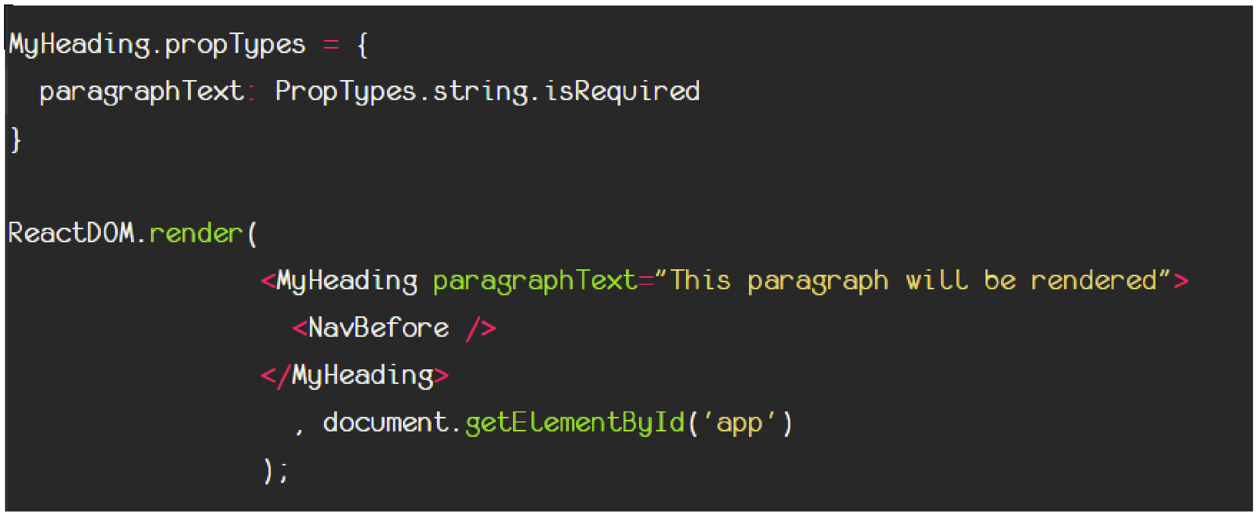
```
jake@LAPTOP-9ULU2A80 MINGW64 ~/Desktop/mondayMorning (tutoring)
$ npm install --save prop-types
```

Dont forget to import it.



```
import PropTypes from 'prop-types';
```

Then in order to use it, you select the component and you set its props to the type you want them to have. The types   can be string, bool, object, number etc. Additionally you can specify if a property will be mandatory on creation of the component or not (required or optional).



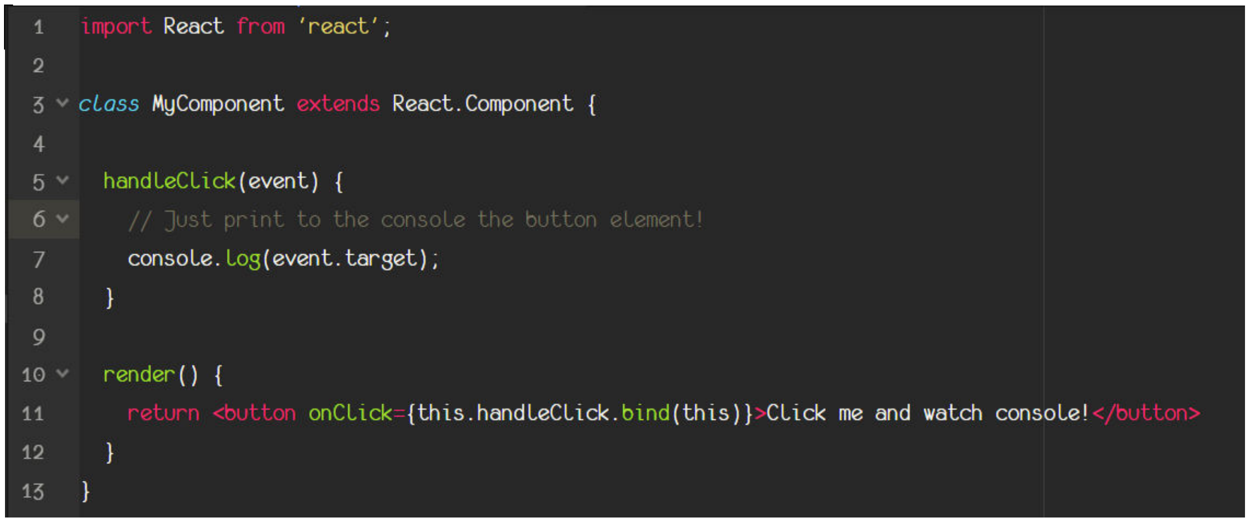
```
MyHeading.propTypes = {
  paragraphText: PropTypes.string.isRequired
}

ReactDOM.render(
                <MyHeading paragraphText="This paragraph will be rendered">
                  <NavBefore />
                </MyHeading>
                  , document.getElementById('app')
                );
```

Creating React events

Events in React can be created through 'inline-style'. Inside the JSX element that triggers the event. That means that there are many attribute names one for every event, some of the most common used are:

onClick
onMouseOver
onChange
onSubmit

The pattern is on and the name of the event together, camelCase syntax! Then the value can be a javascript expression inside {} curly braces. Most of the times you will call a specific function there. Let's see an example:



```
import React from 'react';

class MyComponent extends React.Component {

  handleClick(event) {
    // Just print to the console the button element!
    console.log(event.target);
  }

  render() {
    return <button onClick={this.handleClick.bind(this)}>Click me and watch console!</button>
  }
}
```

The button that is rendered from the component has an event attached to it. When clicked, executes the function with the name handleClick, because this function has not been declared globally but inside the class the this keyword must come before. That indicates that we are talking about a function that is declared inside the same class with the button.

The bind method

If you observe a bit more closely inside the button's event listener you will see something very strange, a 'bind' method attached to the function you have created in the class.

What is this? Is this necessary?

The answer is yes! Normally every function you create inside a component (inside a class) you want to change some properties, variables that are also part of this component. In order to achieve that result you must use the keyword 'this' if you want to gain access to these variables.

The problem is, that anytime you use the keyword 'this' inside the function, you create a referrence NOT to the class, but to the element that triggers the event and eventually the function (that means the button element!!). So if you want to change a variable that has been created inside the class (let’s say with the property’s name, name) you have to go to the function and say

this.name = ‘Jake’;

But the this keyword means to whoever calls the event, so the compiler understands:

button.name = ‘Jake’

DOM elements don’t have such properties, so this will throw an error.

What we can do is to change the point of reference of the this keyword, so when used inside the function, instead of pointing to the button, to point to the class. And that’s exactly what the bind does. It changes the point of reference of the function to the current context (which is the class component).

At the beginning maybe it seems a weird syntax but at the end you will get used to it.

Refs

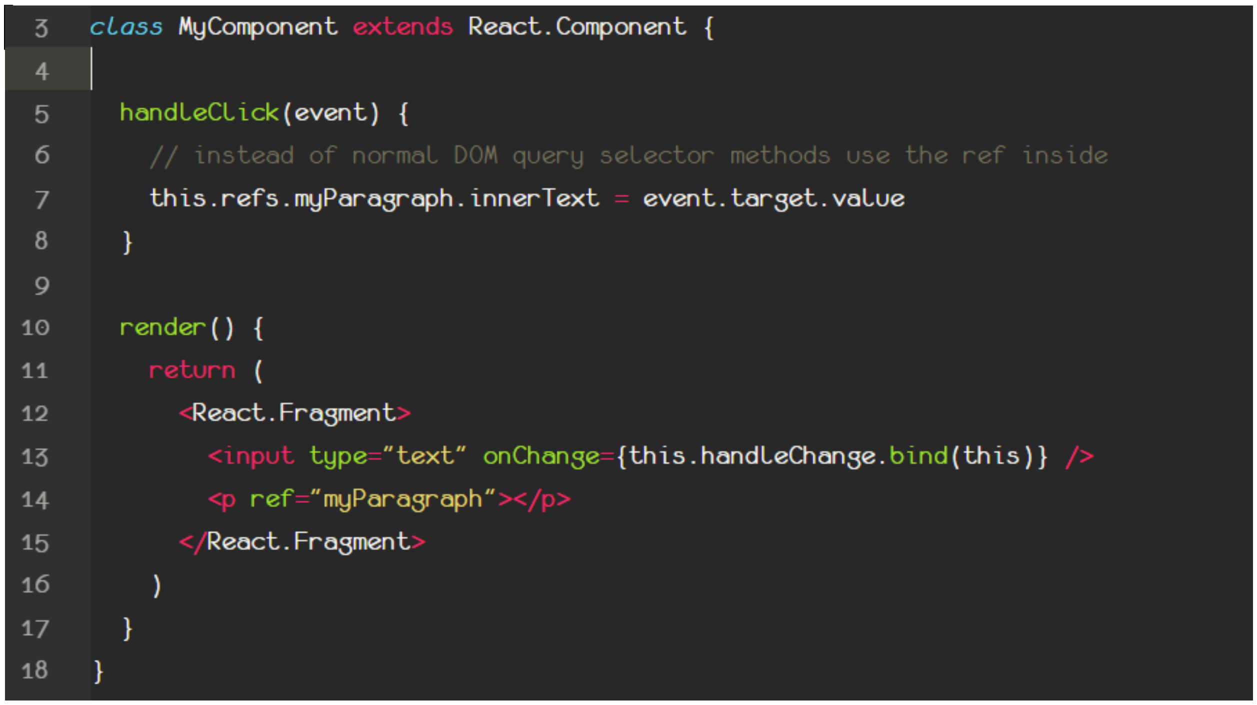
If we want to grab specific JSX elements in React we could still use all the normal methods that VanillaJS give us, like the document.getElementById or the document.querySelector method. But this is not the best or the most efficient way to do it, because these methods start searching for the specified elements across the whole DOM, that means from the beginning <body> tag to the end </body> tag of it.

React provides a way to reference elements only inside the class we are. And that is through the refs object, like seen below.

Every JSX element can have a ref attribute like a special identifier. This is a string. Then inside every function that exists or have been created inside the class, you can grab any JSX element you want by referencing to it.

How? By simply using the refs object that every react component provides. Because every component has a unique refs object, you can have different JSX elements with the same name as ref, as long as they are in different components. But not more than one JSX elements with the same ref inside the same component!

This method is optimized as it limits the range of search only inside the component and to the elements that the render function of the component returns.



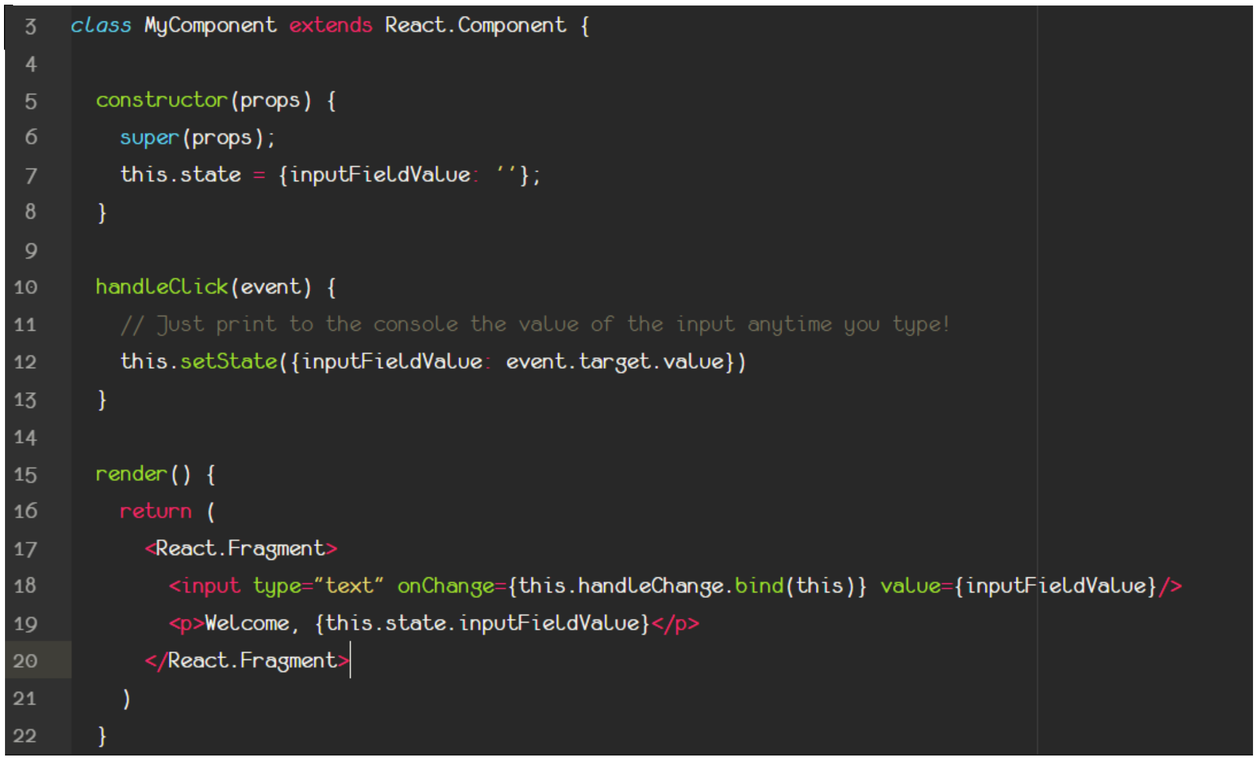
```
class MyComponent extends React.Component {

  handleClick(event) {
    // instead of normal DOM query selector methods use the ref inside
    this.refs.myParagraph.innerText = event.target.value
  }

  render() {
    return (
      <React.Fragment>
        <input type="text" onChange={this.handleChange.bind(this)} />
        <p ref="myParagraph"></p>
      </React.Fragment>
    )
  }
}
```

Create and handle state

Exactly like props every component has a place to store variables and corresponding values. Values that can be used and are accessible inside the component. This object that stores all the variables is called state.

State works exactly like props but with a key difference. State and every property that lies inside, can be changed (so it's what we call mutable). Additionally when the state changes then automatically the render function of the component is called once again.

In our example we will use state to update live the paragraph's content based on what the user types on the input field. We will update the value anytime we have a change of value to the input field form.



```
class MyComponent extends React.Component {

  constructor(props) {
    super(props);
    this.state = {inputFieldValue: ''};
  }

  handleClick(event) {
    // Just print to the console the value of the input anytime you type!
    this.setState({inputFieldValue: event.target.value})
  }

  render() {
    return (
      <React.Fragment>
        <input type="text" onChange={this.handleChange.bind(this)} value={inputFieldValue}/>
        <p>Welcome, {this.state.inputFieldValue}</p>
      </React.Fragment>
    )
  }
```

Anytime you want a live update to happen in react and you don't want to handle all the things manually, you have to create a state object. The state is always created inside the constructor function at the beginning of the class. They way to instantiate a state object is through the:

this.state = {property: value} . This is the pattern that react accepts.

Of course you can have more than one properties that you want react to take a look at and re-render any time something there changes, but for now we will need only one.

So there we create a state property which in the future is going to hold the value of the input. At the beginning we set this equal to empty string. Then we add the value attribute to the input field, so we connect the value of the input field with that place in memory (the state.inputFieldValue).

Now that this connection was made, inside the handleChange function that is executed anytime we change something on the input field, we update the this.state.inputFieldValue.

There is only one way to update the state in React!

That’s through the this.setState function that accepts an object with the properties that you want to change and the corresponding values!

Anytime this function (this.setState) is execute automatically the render function of the component executes itself. This is why you SHOULD NEVER put any function that has a setState inside the render function. Because then you will create an infinite loop and this will make your browser to crash!!

Then the last thing we will do is inside the paragraph instead using any ref, we will put as a content the current state of the inputFieldValue. Any time the state of this property changes, the render function will render the paragraph with the new updated value. Simply as that!!

Components decompose

That’s all good, but for now we have everything inside the same component. The input that holds the functionality for changing the state, and the paragraph that actually uses the state in order to render a property of it.

But then we lose the best thing that react provides. Which is the decomposition of components. That means having your application splitted to many components. But what happens if these components must communicate in order to exchange data?

In our case for example, we could create a completely new component for the paragraph that the only thing that does is to render the paragraph. Wait a second and think. Would this work?

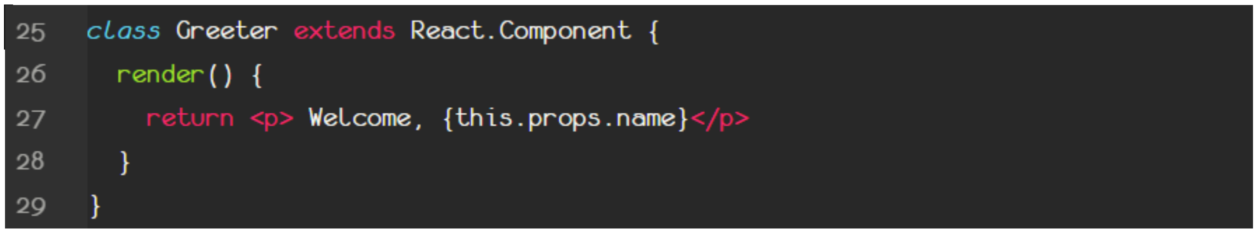
This answer is no! Because now that we have decomposed the component that renders the paragraph, and we still have a reference inside paragraph’s content to the {this.state}. Remember! The new component has no state!!

So how are we going to access the state of the <MyComponent>?

Simple, the answer is through props! We will go inside the render function of the <MyComponent /> and after the button we will render the <Greeter /> component! Then because the greeting component lies inside the <MyComponent /> instead, we will create a new property during rendering named name, and we will make this equal to the state of the <MyComponent />.

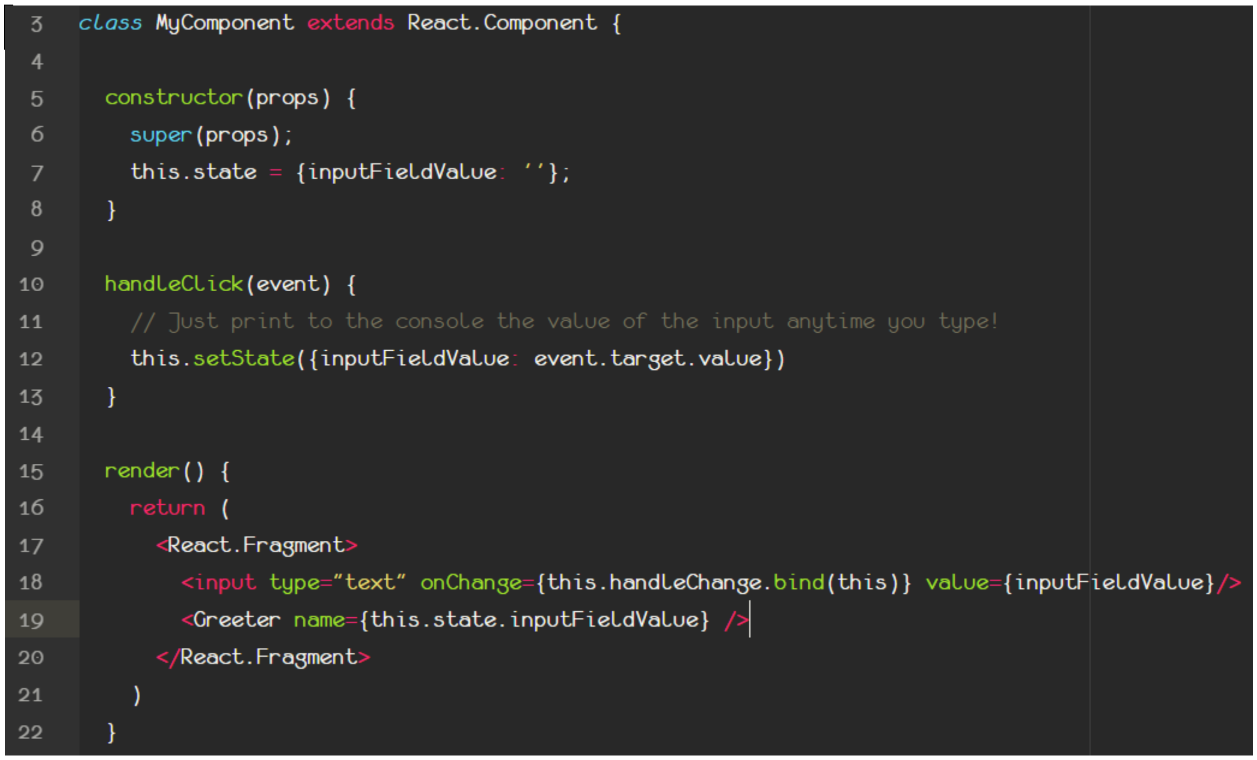
Then inside the <Greeter /> component (that holds the paragraph) instead of using the state, we will use something that paragraph has access to. The props of it's component!!

In the end you are going to end up with something like that:



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

And then, the parent class:



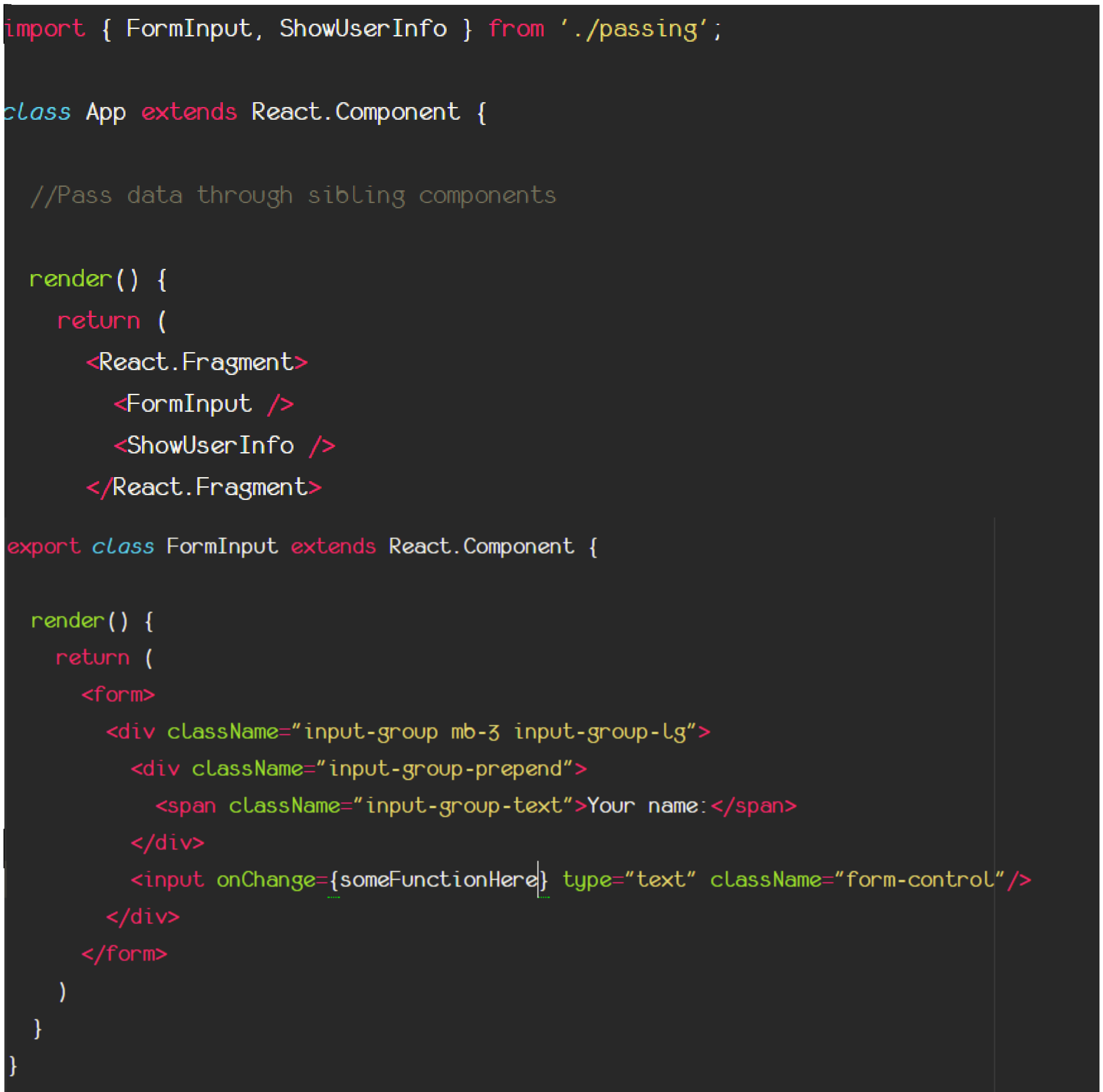
```
class MyComponent extends React.Component {

  constructor(props) {
    super(props);
    this.state = {inputFieldValue: ''};
  }

  handleClick(event) {
    // Just print to the console the value of the input anytime you type!
    this.setState({inputFieldValue: event.target.value})
  }

  render() {
    return (
      <React.Fragment>
        <input type="text" onChange={this.handleChange.bind(this)} value={inputFieldValue}/>
        <Greeter name={this.state.inputFieldValue} />
      </React.Fragment>
    )
  }
```

What if we want to pass data through components who are siblings? And not parent and children? Then the solution is to initialize and maintain state to the parent component of both siblings.

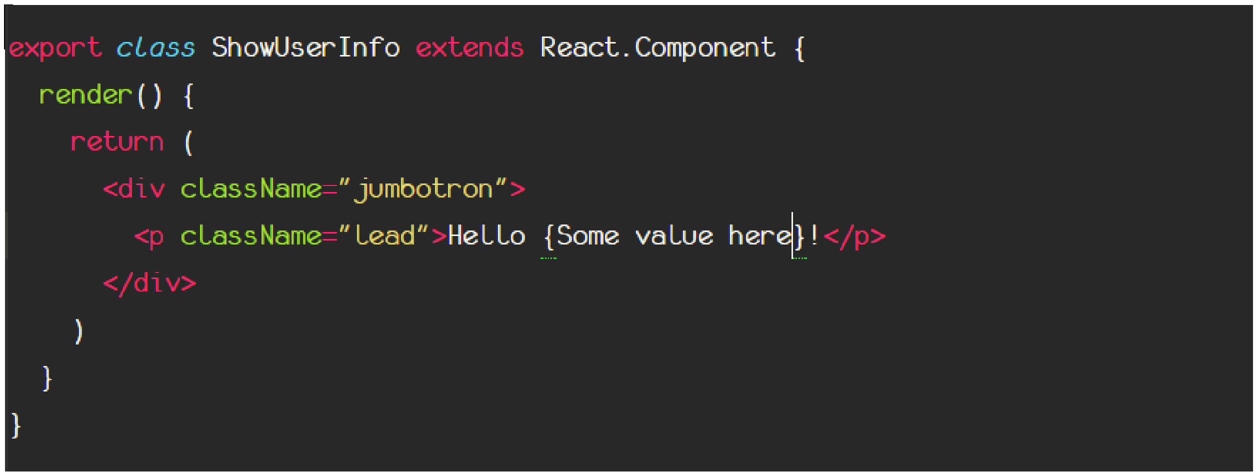
Since they are sibling they should share a parent component! Then they can both have access to the state of the parent component, passed as props.

Already confused? Let’s see an example!

Let’s imagine you have a form with a single input field, and a paragraph that takes the value from this field and displays it. We have seen this before but your structure now has to be the one below:



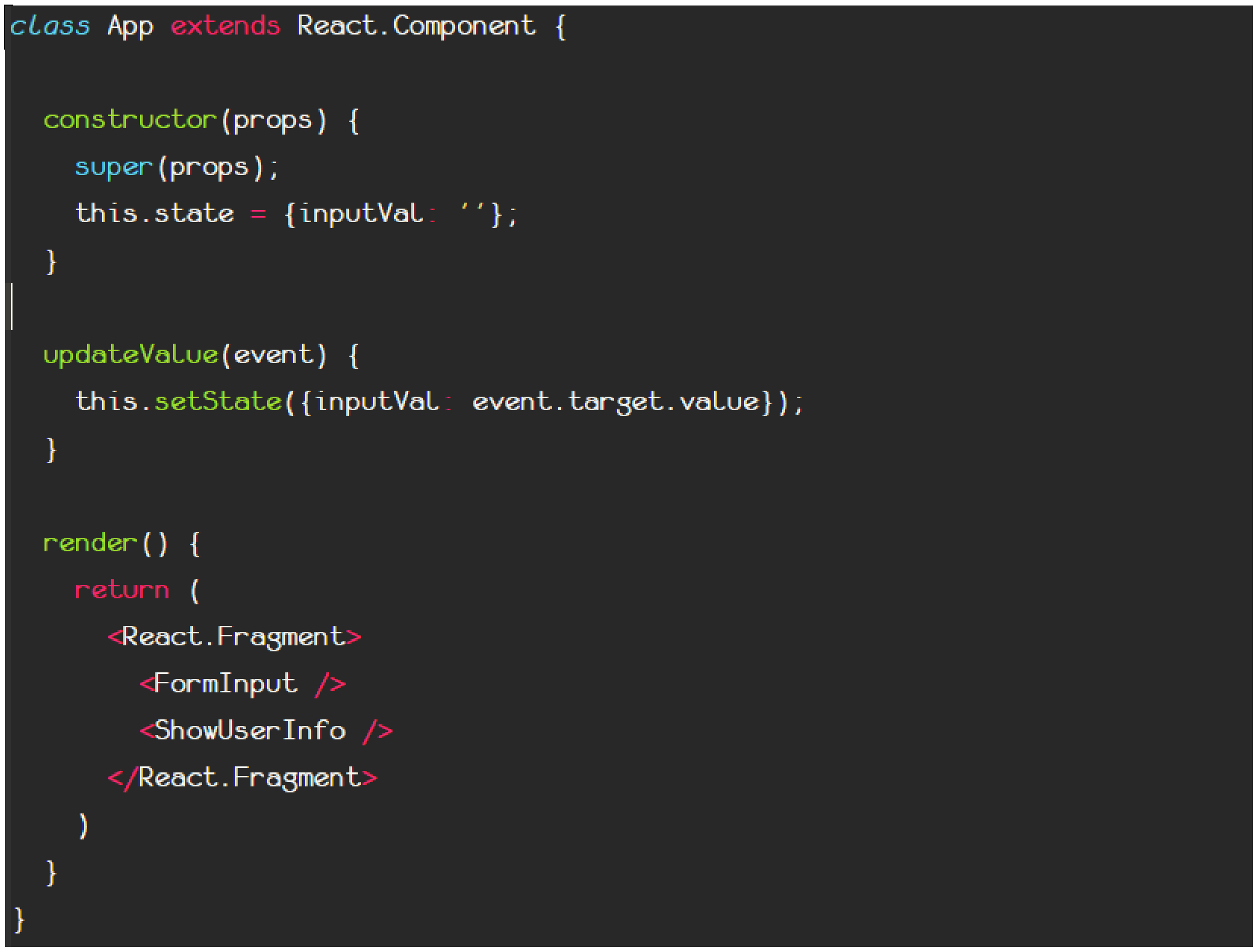
```
import { FormInput, ShowUserInfo } from './passing';

class App extends React.Component {

  //Pass data through sibling components

  render() {
    return (
      <React.Fragment>
        <FormInput />
        <ShowUserInfo />
      </React.Fragment>
export class FormInput extends React.Component {

  render() {
    return (
      <form>
        <div className="input-group mb-3 input-group-lg">
          <div className="input-group-prepend">
            <span className="input-group-text">Your name:</span>
          </div>
          <input onChange={someFunctionHere} type="text" className="form-control"/>
        </div>
      </form>
    )
  }
}
```



```
export class ShowUserInfo extends React.Component {
  render() {
    return (
      <div className=”jumbotron”>
        <p className=”lead”>Hello {Some value here}!</p>
      </div>
    )
  }
}
```

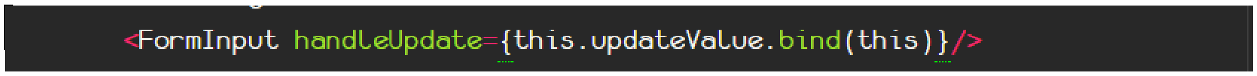
The plan is to initialize and centralize the state somewhere where both components can pull this information from. That’s on the parent component! We will initialize state with the value of the input as an empty string, and then a function that is going to be executed from the form field anytime the data changes there!

Then, anytime the data changes, the value gets updated, the state changes and the paragraph depicts automatically the new value!



```
class App extends React.Component {

  constructor(props) {
    super(props);
    this.state = {inputVal: ‘’};
  }

  updateValue(event) {
    this.setState({inputVal: event.target.value});
  }

  render() {
    return (
      <React.Fragment>
        <FormInput />
        <ShowUserInfo />
      </React.Fragment>
    )
  }
}
```

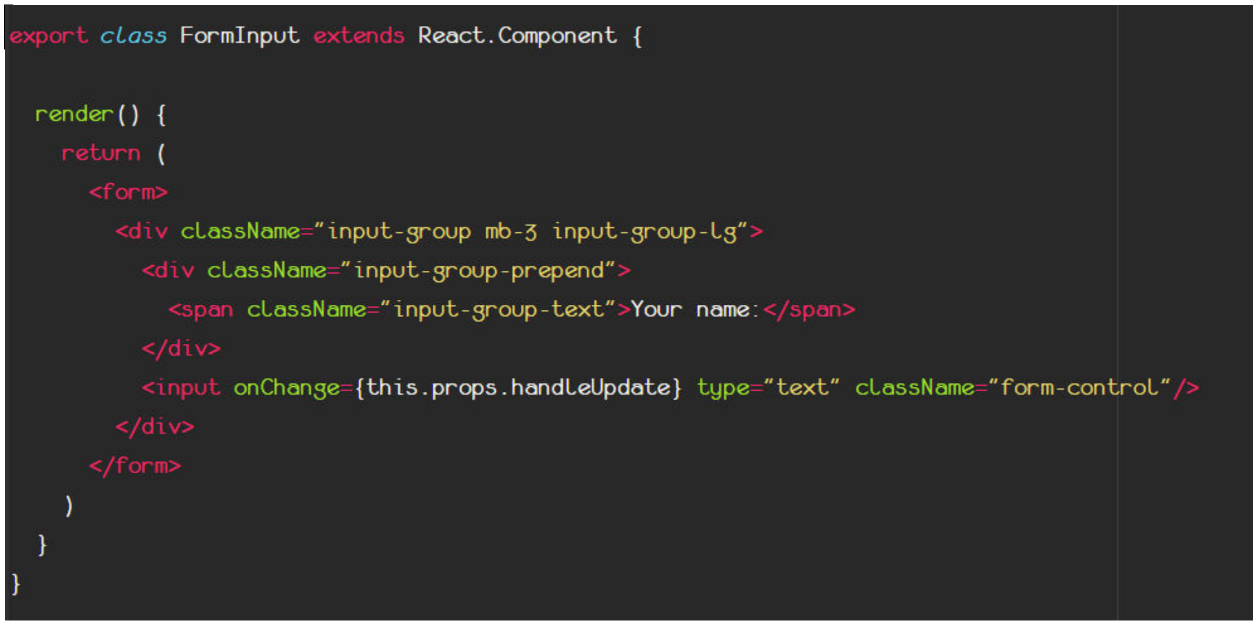
At the same time we will use the current state of the input value every time to update the depicted message! We will use again the props here in order to achieve this effect.



```
        <FormInput handleUpdate={this.updateValue.bind(this)}/>
```

It is extremely important here to bind the function. That means that the function will be executed inside the FormInput component but the state si going to be APP’s component state! So in other words, we change App’s state from the FormInput component.

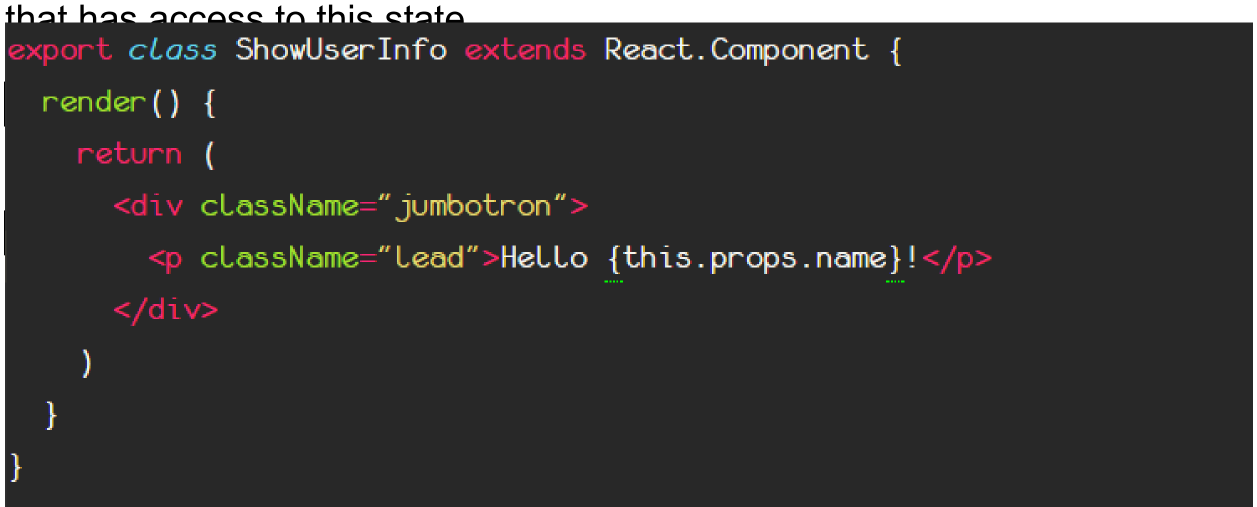
Then we use the passed prop function every time the field is updated and the data is changed!



```
export class FormInput extends React.Component {

  render() {
    return (
      <form>
        <div className=“input-group mb-3 input-group-lg”>
          <div className=“input-group-prepend”>
            <span className=“input-group-text”>Your name:</span>
          </div>
          <input onChange={this.props.handleUpdate} type=“text” className=“form-control”/>
        </div>
      </form>
    )
  }
}
```

Great! Now anytime the value of the input field changes we change the corresponding value in the app state!

Then the next step is to use this value to depict it from another component that has access to this state



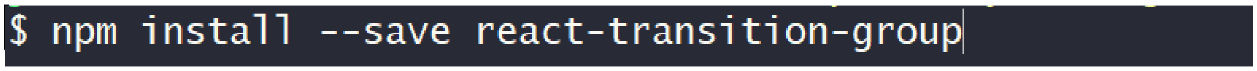
```
export class ShowUserInfo extends React.Component {
  render() {
    return (
      <div className=“jumbotron”>
        <p className=“lead”>Hello {this.props.name}!</p>
      </div>
    )
  }
}
```

You created two siblings components and nested them side by side inside a parent (the app) component, then both have access to the state you created to the parent component. The first has the power to execute the function that changes a specific variable of the state, and the other one depicts this change anytime it happens.

React CSS Transitions

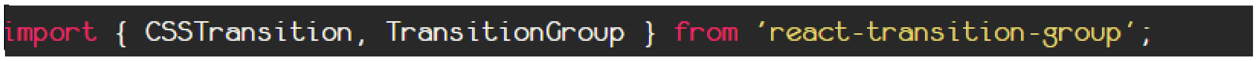
What if we want to perform animations or transitions to JSX elements? There are react-libraries that help us performing these tasks. For exampling we can perform a transition or an animation, exactly the same way, we would do it in css, when something mounts to or unmounts from the DOM.

We use the ‘react-transition-group’ in order to achieve this effect. First we need to install it:



```
$ npm install --save react-transition-group
```

Then we import it to our project:

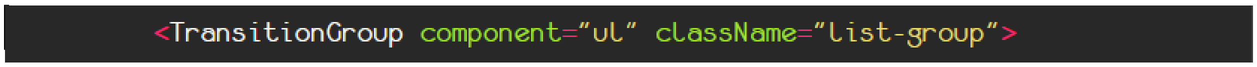


```
import { CSSTransition, TransitionGroup } from ‘react-transition-group’;
```

Specifically we need two different components, the CSSTransition component which can perform a transition or animation to a solo element, and the TransitionGroup component which can handle the same task but for a group of elements.

In our case we are going to use both since we are going to fade in and fade out all the <li> items we have inside the our <ul>.

The first thing we want to do is to identify the elements we want to apply the transition and replace the parent element of these (in our case the <ul> element) with the <TransitionGroup /> component. This semantically is for the DOM an empty component that just notifies React that all the children of this can be transitioned when mounting or unmounting from the DOM.

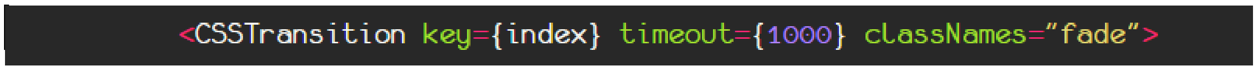


```
        <TransitionGroup component=”ul” className=”list-group”>
```

By passing the “ul” to the component prop, it keeps the <TransitionGroup >

component from a meaningless for the DOM component to an actual <ul> element with a class of “list-group”. This acts as a wrapper to all the elements that will be individually transitioned after.

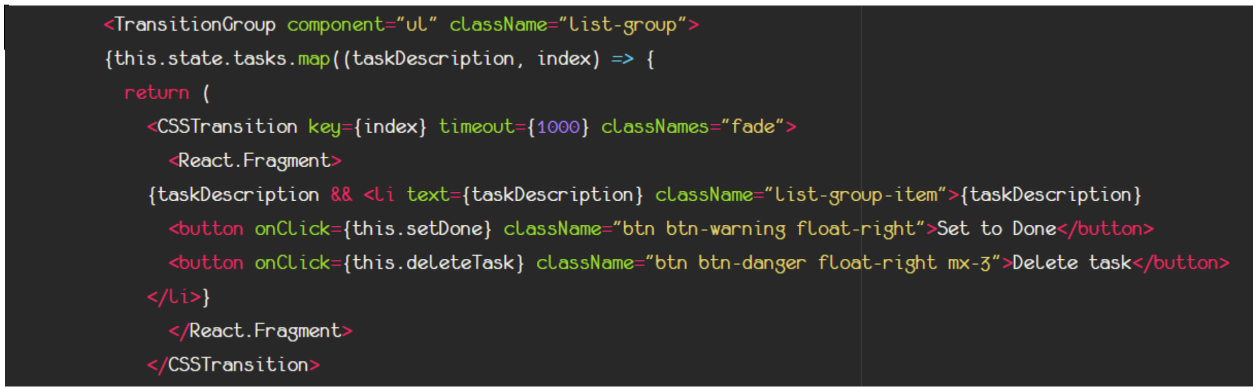
Then we need to apply transition to each element. Inside the map we put:



```
        <CSSTransition key={index} timeout={1000} classNames=“fade”>
```

We put the key property because the CSSTransition component is the first outer element of the map function. The timeout specifies after how many milliseconds each <li> will be mounted and unmounted to and by the DOM. Last but not least we give a className to the **classNames** attribute.

Overall the big picture will look like this:



```
      <TransitionGroup component=“ul” className=“list-group”>
      {this.state.tasks.map((taskDescription, index) => {
        return (
          <CSSTransition key={index} timeout={1000} classNames=“fade”>
            <React.Fragment>
          {taskDescription && <li text={taskDescription} className=“list-group-item”>{taskDescription}
            <button onClick={this.setDone} className=“btn btn-warning float-right”>Set to Done</button>
            <button onClick={this.deleteTask} className=“btn btn-danger float-right mx-3”>Delete task</button>
          </li>}
            </React.Fragment>
          </CSSTransition>
```

Don’t forget to close appropriately the corresponding tags for the CSSTransition and the TransitionGroup components. Like the picture shown above.

Then the only thing we must do is to configure these 4 classes, that represent the 4 different stages of an element. Take notice that we have named our class in our CSSTransition component as ‘fade’. This name could be anything but from this name, 4 different classes will be created for every li now:

.fade-enter => When the element is about to mount to the DOM

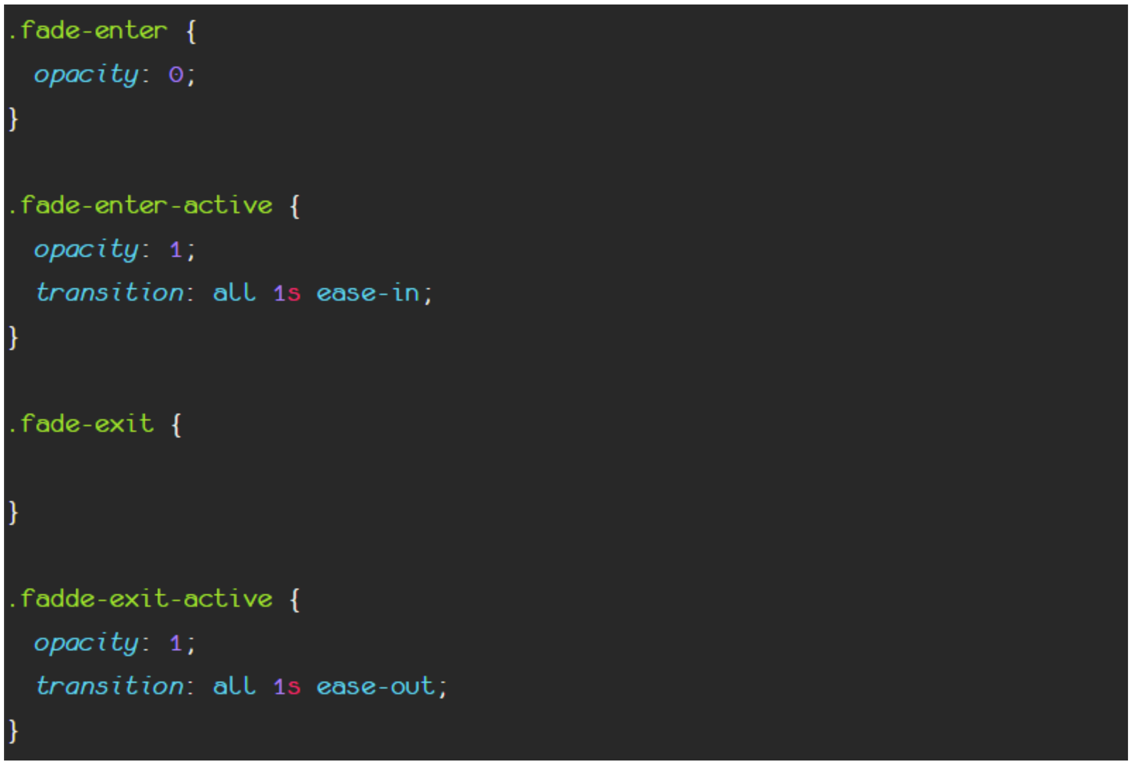
.fade-enter-active => When the element is has mounted to the DOM

The duration between .fade-enter and .fade-enter-active is going to be the timeout duration you have defined there. Be careful! This is not the transition or animation duration, this will be defined through css inside the appropriate classes! This is the amount of time, react is going to wait in order to mount an element to the DOM.The same way for unmounting, two new classes exist:

.fade-exit => When the element is about to unmount from DOM

.fade-exit-active => When the elements was succesfully unmounted from DOM.

Now we just configure this classes in order to achieve the transition effect!Take care, that transition takes place at the end of each phase and not the beginning as normal css!!! You initialize the opacity to 0, and then you transition the opacity to 1 after the end of each phase. Normally the duration must be the same with the timeout. At the end we end up with something like that:



```
.fade-enter {
  opacity: 0;
}

.fade-enter-active {
  opacity: 1;
  transition: all 1s ease-in;
}

.fade-exit {


}

.fadde-exit-active {
  opacity: 1;
  transition: all 1s ease-out;
}
```

Everything should work now, and everytime an <li> is created on the fly instead of just popping itself to the DOM is faded in. On exit (on deleted) is faded-out. The transition lasts for one second, and after the second React mounts and unmounts.

React Routing and SPA’s

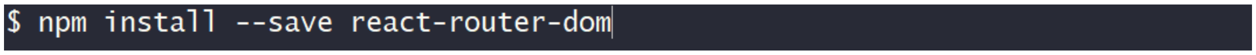
Until know everything we have seen serves a specific architecture model. Until know we were creating html files, and we were navigating from one html file to the next one through links (anchor elements). This style of software architecture is a ‘multi-page-application’. Any time we want to navigate into a different file, we make an external request to the server and the server responds by serving another static html file. Although this seemed to work quite well until know, we put a lot of load to our server’s shoulder. Normally we would like to reduce some of this load.

Here comes another style to the rescue. The Single Page Applications or SPA’s. The main idea behind this architecture is that the server serves only one html file to the client which contains all different react components and versions (we call them ‘routes’) of the same file. Then we put route links instead of normal links that change the route url in the client. Then we specify which component (or JSX elements) are going to be displayed under this url route. This technique has a tremendous advantage since we don’t need to make an external request to the server any time we navigate. Everything happens on the client (browser) side and we don’t need to reload the page. So everything is faster.

Although routing is a powerful feature that comes with almost every client-side js framework this days, there is also a con. In order to serve only one html file and include these different versions (routes) that the users is going to probably need during his navigation to our site, we need to fetch everything within the original request. That means that the original request to the server for the one and only html file (the index.html probably) is going to contain all the possible routes that the user is going to use or not. Thus the initial request typically is going to be served slower than a MPA (multi-page app).

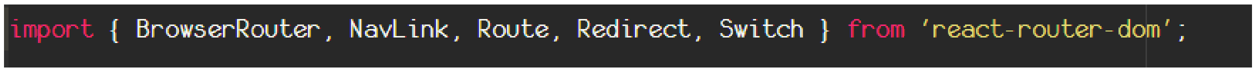
After that obstacle is surpassed, of course everything runs smoothly within the browser enviroment. External requests can additionally made to the server (AJAX calls etc) but only to fetch and exchange data (mostly to JSON format) and not whole HTML files. Let’s take a look on how we can implement such a functionality.

First we need to install the react-router-dom library.



```
$ npm install --save react-router-dom
```

We install it as a production dependency. Then we need to import couple of Components. As shown below:



```
import { BrowserRouter, NavLink, Route, Redirect, Switch } from 'react-router-dom';
```

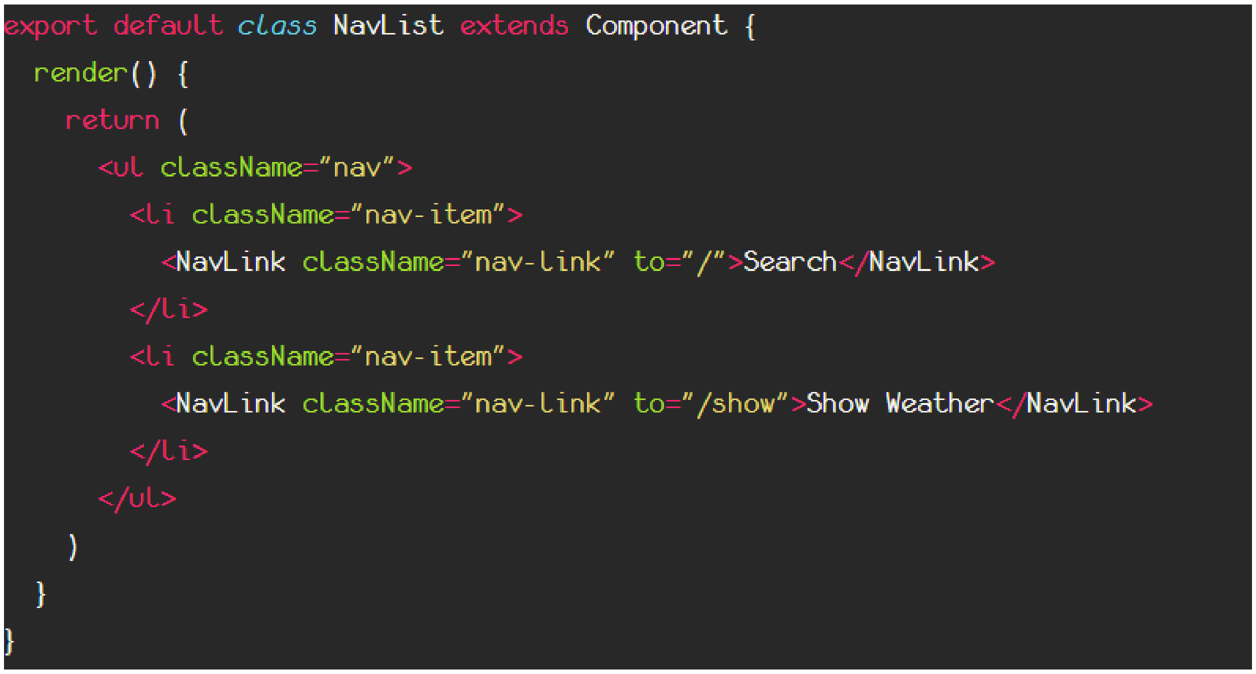
We don’t need necessarily all these components always. But for sure we are going to need the first three.

<BrowserRouter> (or the <HashRouter for some purposes) is the parent element. Anytime we want to implement different routes, we need to nest them inside a <BrowserRouter> component.

The <NavLink> component is equivalent to the <a>. Instead of reloading and fetching a new html page, changes the url navigation to the given attribute (through the “to” attribute).

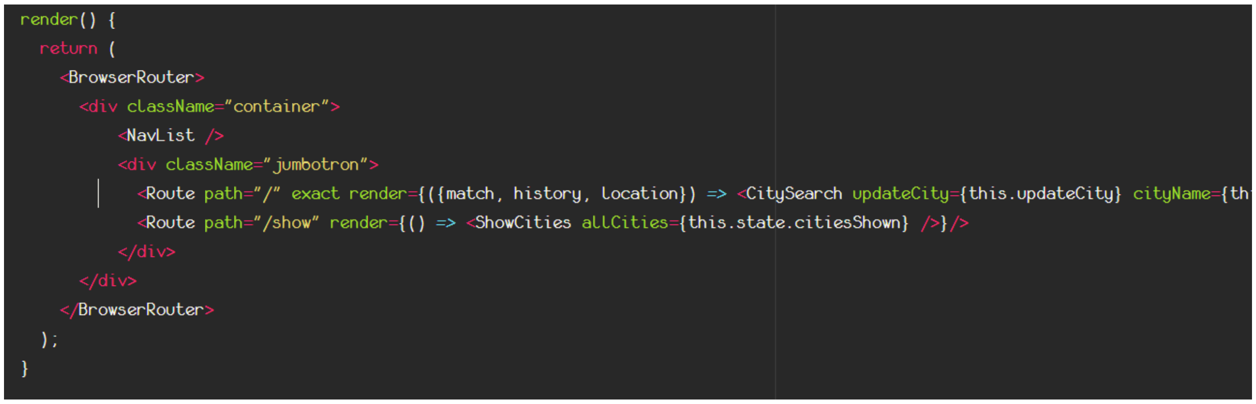
The <Route> indiciates which component (or JSX elements) is going to be shown in a given route. This is specified through the path attribute and the component (or render for JSX) atttributes, respectively.

As you can see below we have three routing links which change the url t respective addresses. We use the NavLink component and the to attribute to indicate where this click of this link should get the user.



```
export default class NavList extends Component {
  render() {
    return (
      <ul className="nav">
        <li className="nav-item">
          <NavLink className="nav-link" to="/">Search</NavLink>
        </li>
        <li className="nav-item">
          <NavLink className="nav-link" to="/show">Show Weather</NavLink>
        </li>
      </ul>
    )
  }
}
```

Then we have a link that, when clicked, gets us to the home/initial page. And a second link to a /show route. Now we need to decide what to depict when we navigate ourselves there. As said before, we implement the actual routes as shown below:



```
render() {
  return (
    <BrowserRouter>
      <div className="container">
          <NavList />
          <div className="jumbotron">
            <Route path="/" exact render={({match, history, location}) => <CitySearch updateCity={this.updateCity} cityName={th
            <Route path="/show" render={() => <ShowCities allCities={this.state.citiesShown} />}/>
          </div>
      </div>
    </BrowserRouter>
  );
}
```

In our case when the first link is clicked we show (via the render method) the <CitySearch /> component (don’t worry about the parameters inside the arrow function for now).

When the second link is cliked (we navigate to the /show) we bring to life the <ShowCities /> component. You can also use the component attribute instead render. Although here we need to pass also some props inside every component, and we use the render method, that accepts an arrow function. This function then renders the component or pure JSX if you want.

The redirect component

What if we want to redirect the user programmatically? Without a click event on a link? Well, there are couple of ways to do that. The one is through pushing a new url path to history. But for now we are going to use the <Redirect /> component we imported above. Then we place the component inside some render function and everytime this render function is executed we redirect the user to a new url.

```
<Redirect to=”/home” />
```

The code above when executed will redirect the user to the /home.

The Switch component

Sometimes we want to show only one route from a group of routes. Of course we can specify the “exact” path and thus ensure that no route can be depicted accidentally as a partial match, but what if we want to group routes and select to show the ONLY the first match. Then we can use the <Switch> component and nest routes inside of it. The nested routes are going to be evaluated one by one, and once there is a match, it will be the selected route. Then the evaluation of the other remaining routes stops.

```
<Switch>
  <Route to=”/” component={ExampleOne} />
  <Route to=”/home” component={ExampleTwo} />
  <Route to=”/different” render={() => <ExampleThree />} />
</Switch>
```

In this example without having the switch, anytime we were either on the /home or on the /different we would render the corresponding component AND the <ExampleOne /> as well (due to partial matching). With the Switch component we render ONLY the first match.

Be careful! In our case, and because we have no exact keyword, regardless of which one of the three routes is the active route (the current url), we will depict always the <ExampleOne /> component.

Why?   Because since these three routes are inside a switch keyword. The first match will be depicted. Regardless if we have clicked on the link that gets us to just /, or to /home, or /different. The first (with just a /), evaluates to true due to partial match. (maybe you want to put an exact keyword there to avoid this issue). The others will not be evaluated at all!

Routing and SPA's are a powerful feature of client-side architecture, and although we covered some basics, it is highly recommended to look for further resources and topics, like dynamic routes, params, the match, history and location objects etc.

If you want to learn more, you can start the journey here:

https://www.kirupa.com/react/creating_single_page_app_react_using_react_router.htm