What is React:

React is a JavaScript library for building user interfaces.

React is used to build single page applications.

React allows us to create reusable UI components.

Learning by Examples

Our "Show React" tool makes it easy to demonstrate React, it shows both the code and the result.

Example:

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

class Test extends React.Component {
  render() {
    return <h1>Hello World!</h1>;
  }
}

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

Create React App

In order to learn and test React, you should set up a React Environment on your computer.

This tutorial uses the `create-react-app`.

The `create-react-app` is an officially supported way to create React applications.

If you have NPM and Node.js installed, you can create a React application by first installing the create-react-app.

Install create-react-app by running this command in your terminal:

```
C:\Users\Your Name>npm install -g create-react-app
```

You are now ready to create your first React application!

Run this command to create a React application named `myfirstreact`:

```
C:\Users\Your Name>npx create-react-app myfirstreact
```

The `create-react-app` will set up everything you need to run a React application.

Note: This tutorial uses `create-react-app` to demonstrate React examples. You will not be able to run the same examples on your computer if you do not install the `create-react-app` environment.

Run the React Application

If you followed the two commands above, you are ready to run your first *real* React application!

Run this command to move to the `myfirstreact` directory:

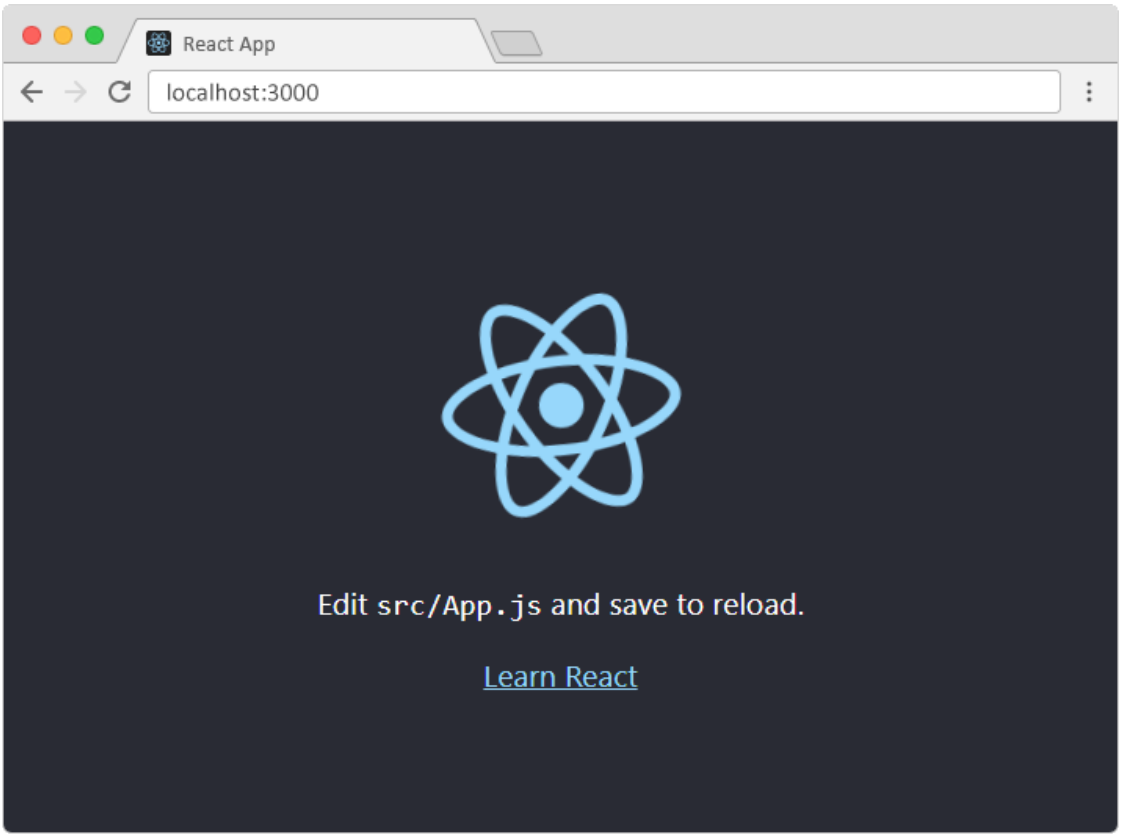
```
C:\Users\Your Name>cd myfirstreact
```

Run this command to execute the React application `myfirstreact`:

```
C:\Users\Your Name\myfirstreact>npm start
```

A new browser window will pop up with your newly created React App! If not, open your browser and type `localhost:3000` in the address bar.

The result:



What is React?

React is a JavaScript library created by Facebook.

React is a tool for building UI components.

How does React Work?

React creates a VIRTUAL DOM in memory.

Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM.

React only changes what needs to be changed!

React finds out what changes have been made, and changes **only** what needs to be changed.

You will learn the various aspects of how React does this in the rest of this tutorial.

React.JS History

Current version of React.JS is V16.8.6 (March 2019).

Initial Release to the Public (V0.3.0) was in July 2013.

React.JS was first used in 2011 for Facebook's Newsfeed feature.

Facebook Software Engineer, Jordan Walke, created it.

The create-react-app version 2.0 package was released in October 2018.

Create-react-app version 2.0 supports Babel 7, webpack 4, and Jest23.

What is ES6?

ES6 stands for ECMAScript 6.

ECMAScript was created to standardize JavaScript, and ES6 is the 6th version of ECMAScript, it was published in 2015, and is also known as ECMAScript 2015.

Why Should I Learn ES6?

React uses ES6, and you should be familiar with some of the new features like:

- Classes
- Arrow Functions
- Variables (let, const, var)

Classes

ES6 introduced classes.

A class is a type of function, but instead of using the keyword `function` to initiate it, we use the keyword `class`, and the properties is assigned inside a`constructor()` method.

Example

A simple class constructor:

```
class Car {
  constructor(name) {
    this.brand = name;
  }
}
```

Now you can create objects using the Car class:

Example

Create an object called "mycar" based on the Car class:

```
class Car {
  constructor(name) {
    this.brand = name;
  }
}

mycar = new Car("Ford");
```

Method in Classes

You can add your own methods in a class:

Example

Create a method named "present":

```
class Car {
  constructor(name) {
    this.brand = name;
  }

  present() {
    return 'I have a ' + this.brand;
```

```
  }
}

mycar = new Car("Ford");
mycar.present();
```

As you can see in the example above, you call the method by referring to the object's method name followed by parentheses (parameters would go inside the parentheses).

Class Inheritance

To create a class inheritance, use the `extends` keyword.

A class created with a class inheritance inherits all the methods from another class:

Example

Create a class named "Model" which will inherit the methods from the "Car" class:

```
class Car {
  constructor(name) {
    this.brand = name;
  }

  present() {
    return 'I have a ' + this.brand;
  }
}

class Model extends Car {
  constructor(name, mod) {
    super(name);
    this.model = mod;
  }
  show() {
      return this.present() + ', it is a ' + this.model
```

```
  }
}
mycar = new Model("Ford", "Mustang");
mycar.show();
```

The `super()` method refers to the parent class.

By calling the `super()` method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.

Arrow Functions

Arrow functions were introduced in ES6.

Arrow functions allow us to write shorter function syntax:

Before:

```
hello = function() {
  return "Hello World!";
}
```

With Arrow Function:

```
hello = () => {
  return "Hello World!";
}
```

It gets shorter! If the function has only one statement, and the statement returns a value, you can remove the brackets *and* the `return` keyword:

Arrow Functions Return Value by Default:

```
hello = () => "Hello World!";
```

Note: This works only if the function has only one statement.

If you have parameters, you pass them inside the parentheses:

Arrow Function With Parameters:

```
hello = (val) => "Hello " + val;
```

In fact, if you have only one parameter, you can skip the parentheses as well:

Arrow Function Without Parentheses:

```
hello = val => "Hello " + val;
```

What About `this`?

The handling of `this` is also different in arrow functions compared to regular functions.

In short, with arrow functions there are no binding of `this`.

In regular functions the `this` keyword represented the object that called the function, which could be the window, the document, a button or whatever.

With arrow functions the `this` keyword *always* represents the object that defined the arrow function.

Let us take a look at two examples to understand the difference.

Both examples call a method twice, first when the page loads, and once again when the user clicks a button.

The first example uses a regular function, and the second example uses an arrow function.

The result shows that the first example returns two different objects (window and button), and the second example returns the Header object twice.

Example

With a regular function `this` represents the object that called the function:

```
class Header {
  constructor() {
    this.color = "Red";
  }
```

```
//Regular function:
  changeColor = function() {
    document.getElementById("demo").innerHTML += this;
  }
}

myheader = new Header();

//The window object calls the function:
window.addEventListener("load", myheader.changeColor);

//A button object calls the function:
document.getElementById("btn").addEventListener("click",
myheader.changeColor);
```

Example

With an arrow function `this` represents the Header object no matter who called the function:

```
class Header {
  constructor() {
    this.color = "Red";
  }

//Arrow function:
  changeColor = () => {
    document.getElementById("demo").innerHTML += this;
  }
}

myheader = new Header();



//The window object calls the function:
window.addEventListener("load", myheader.changeColor);
```

```
//A button object calls the function:
document.getElementById("btn").addEventListener("click",
myheader.changeColor);
```

Remember these differences when you are working with functions. Sometimes the behavior of regular functions is what you want, if not, use arrow functions.

Variables

Before ES6 there were only one way of defining your variables: with the `var` keyword. If you did not define them, they would be assigned to the global object. Unless you were in strict mode, then you would get an error if your variables were undefined.

Now, with ES6, there are three ways of defining your variables: `var`, `let`, and `const`.

var

```
var x = 5.6;
```

If you use `var` outside of a function, it belongs to the global scope.

If you use `var` inside of a function, it belongs to that function.

If you use `var` inside of a block, i.e. a for loop, the variable is still available outside of that block.

`var` has a *function* scope, not a *block* scope.

let

```
let x = 5.6;
```

`let` has a *block* scope.

`let` is the block scoped version of `var`, and is limited to the block (or expression) where it is defined.

If you use `let` inside of a block, i.e. a for loop, the variable is only available inside of that loop.

const

```
const x = 5.6;
```

`const` is a variable that once it has been created, its value can never change.

`const` has a *block* scope.

What is JSX?

JSX stands for JavaScript XML.

JSX allows us to write HTML in React.

JSX makes it easier to write and add HTML in React.

Coding JSX

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any `createElement()` and/or `appendChild()` methods.

JSX converts HTML tags into react elements.

You are not required to use JSX, but JSX makes it easier to write React applications.

Let us demonstrate with two examples, the first uses JSX and the second does not:

Example 1

JSX:

```
const myelement = <h1>I Love JSX!</h1>;


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

Example 2

Without JSX:

```
const myelement = React.createElement('h1', {}, 'I do not use JSX!');


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

As you can see in the first example, JSX allows us to write HTML directly within the JavaScript code.

JSX is an extension of the JavaScript language based on ES6, and is translated into regular JavaScript at runtime.

Expressions in JSX

With JSX you can write expressions inside curly braces `{ }`.

The expression can be a React variable, or property, or any other valid JavaScript expression. JSX will execute the expression and return the result:

Example

Execute the expression `5 + 5`:

```
const myelement = <h1>React is {5 + 5} times better with JSX</h1>;
```

Inserting a Large Block of HTML

To write HTML on multiple lines, put the HTML inside parentheses:

Example

Create a list with three list items:

```
const myelement = (
  <ul>
    <li>Apples</li>
    <li>Bananas</li>
```

```
    <li>Cherries</li>
  </ul>
);
```

One Top Level Element

The HTML code must be wrapped in ONE top level element.

So if you like to write *two* headers, you must put them inside a parent element, like a `div` element

Example

Wrap two headers inside one DIV element:

```
const myelement = (
  <div>
    <h1>I am a Header.</h1>
    <h1>I am a Header too.</h1>
  </div>
);
```

JSX will throw an error if the HTML is not correct, or if the HTML misses a parent element.

Elements Must be Closed

JSX follows XML rules, and therefore HTML elements must be properly closed.

Example

Close empty elements with `/>`

```
const myelement = <input type="text" />;
```

JSX will throw an error if the HTML is not properly closed.

React Components

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and returns HTML via a render function.

Components come in two types, Class components and Function components, in this tutorial we will concentrate on Class components.

Create a Class Component

When creating a React component, the component's name must start with an upper case letter.

The component has to include the `extend React.Component` statement, this statement creates an inheritance to React.Component, and gives your component access to React.Component's functions.

The component also requires a `render()` method, this method returns HTML.

Example

Create a Class component called `Car`

```
class Car extends React.Component {
  render() {
    return <h2>Hi, I am a Car!</h2>;
  }
}
```

Now your React application has a component called Car, which returns a `<h2>` element.

To use this component in your application, use similar syntax as normal HTML: `<Car />`

Example

Display the Car component in the "root" element:

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

Create a Function Component

Here is the same example as above, but created using a Function component instead.

A Function component also returns HTML, and behaves pretty much the same way as a Class component, but Class components have some additions, and will be preferred in this tutorial.

Example

Create a Function component called Car

```
function Car() {
  return <h2>Hi, I am also a Car!</h2>;
}
```

Once again your React application has a Car component.

Refer to the Car component as normal HTML (except in React, components *must* start with an upper case letter):

Example

Display the Car component in the "root" element:

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

Component Constructor

If there is a `constructor()` function in your component, this function will be called when the component gets initiated.

The constructor function is where you initiate the component's properties.

In React, component properties should be kept in an object called `state`.

You will learn more about `state` later in this tutorial.

The constructor function is also where you honor the inheritance of the parent component by including the `super()` statement, which executes the parent component's constructor function, and your component has access to all the functions of the parent component (`React.Component`).

Example

Create a constructor function in the Car component, and add a color property:

```
class Car extends React.Component {
  constructor() {
    super();
    this.state = {color: "red"};
  }
  render() {
    return <h2>I am a Car!</h2>;
  }
}
```

Use the color property in the render() function:

Example

```
class Car extends React.Component {
  constructor() {
    super();
    this.state = {color: "red"};
  }
  render() {
    return <h2>I am a {this.state.color} Car!</h2>;
```

```
  }
}
```

Props

Another way of handling component properties is by using `props`.

Props are like function arguments, and you send them into the component as attributes.

You will learn more about `props` in the next chapter.

Example

Use an attribute to pass a color to the Car component, and use it in the render() function:

```
class Car extends React.Component {
  render() {
    return <h2>I am a {this.props.color} Car!</h2>;
  }
}
ReactDOM.render(<Car color="red"/>, document.getElementById('root'));
```

Components in Components

We can refer to components inside other components:

Example

Use the Car component inside the Garage component:

```
class Car extends React.Component {
  render() {
    return <h2>I am a Car!</h2>;
```

```
  }
}

class Garage extends React.Component {
  render() {
    return (
      <div>
      <h1>Who lives in my Garage?</h1>
      <Car />
      </div>
    );
  }
}

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

Components in Files

React is all about re-using code, and it can be smart to insert some of your components in separate files.

To do that, create a new file with a `.js` file extension and put the code inside it:

Note that the file must start by importing React (as before) and it has to end with the statement `export default Car;`.

Example

This is the new file, we named it "App.js":

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

class Car extends React.Component {
  render() {
    return <h2>Hi, I am a Car!</h2>;
```

```
}
}


export default Car;
```

To be able to use the Car component, you have to import the file in your application.

Example

Now we import the "App.js" file in the application, and we can use the Car component as if it was created here.

```
import React from 'react';
import ReactDOM from 'react-dom';
import Car from './App.js';
ReactDOM.render(<Car />, document.getElementById('root'));
```

React Props

Props are arguments passed into React components.

Props are passed to components via HTML attributes.

React Props

React Props are like function arguments in JavaScript *and* attributes in HTML.

To send props into a component, use the same syntax as HTML attributes:

Example

Add a "brand" attribute to the Car element:

```
const myelement = <Car brand="Ford" />;
```

The component receives the argument as a `props` object:

Example

Use the brand attribute in the component:

```
class Car extends React.Component {
  render() {
    return <h2>I am a {this.props.brand}!</h1>;
  }
}
```

Pass Data

Props are also how you pass data from one component to another, as parameters.

Example

Send the "brand" property from the Garage component to the Car component:

```
class Car extends React.Component {
  render() {
    return <h2>I am a {this.props.brand}!</h2>;
  }
}

class Garage extends React.Component {
  render() {
    return (
      <div>
      <h1>Who lives in my garage?</h1>
      <Car brand="Ford" />
      </div>
    );
  }
```

```
}

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

If you have a variable to send, and not a string as in the example above, you just put the variable name inside curly brackets:

Example

Create a variable named "carname" and send it to the Car component:

```
class Car extends React.Component {
  render() {
    return <h2>I am a {this.props.brand}!</h2>;
  }
}

class Garage extends React.Component {
  render() {
    const carname = "Ford";
    return (
      <div>
      <h1>Who lives in my garage?</h1>
      <Car brand={carname} />
      </div>
    );
  }
}

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

Or if it was an object:

Example

Create an object named "carinfo" and send it to the Car component:

```
class Car extends React.Component {
```

```
  render() {
    return <h2>I am a {this.props.brand.model}!</h2>;
  }
}

class Garage extends React.Component {
  render() {
    const carinfo = {name: "Ford", model: "Mustang"};
    return (
      <div>
      <h1>Who lives in my garage?</h1>
      <Car brand={carinfo} />
      </div>
    );
  }
}

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

Props in the Constructor

If your component has a constructor function, the props should always be passed to the constructor and also to the React.Component via the `super()` method.

Example

```
class Car extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return <h2>I am a Car!</h2>;
  }
}
```

```
ReactDOM.render(<Car model="Mustang"/>,
document.getElementById('root'));
```

Note: React Props are read-only! You will get an error if you try to change their value.

React State

React components has a built-in `state` object.

The `state` object is where you store property values that belongs to the component.

When the `state` object changes, the component re-renders.

Creating the `state` Object

The `state` object is initialized in the constructor:

Example:

Specifiy the `state` object in the constructor method:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {brand: "Ford"};
  }
  render() {
    return (
      <div>
        <h1>My Car</h1>
      </div>
```

```
    );
  }
}
```

The `state` object can contain as many properties as you like:

Example:

Specify all the properties your component need:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My Car</h1>
      </div>
    );
  }
}
```

Using the `state` Object

Refer to the `state` object anywhere in the component by using the `this.state.propertyname` syntax:

Example:

Refer to the `state` object in the `render()` method:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
          {this.state.model}
          from {this.state.year}.
        </p>
      </div>
    );
  }
}
```

Changing the `state` Object

To change a value in the state object, use the `this.setState()` method.

When a value in the `state` object changes, the component will re-render, meaning that the output will change according to the new value(s).

Example:

Add a button with an `onClick` event that will change the color property:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  changeColor = () => {
    this.setState({color: "blue"});
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
          {this.state.model}
          from {this.state.year}.
        </p>
        <button
          type="button"
          onClick={this.changeColor}
        >Change color</button>
```

```
      </div>
    );
  }
}
```

React Events

Just like HTML, React can perform actions based on user events.

React has the same events as HTML: click, change, mouseover etc.

Adding Events

React events are written in camelCase syntax:

`onClick` instead of `onclick`.

React event handlers are written inside curly braces:

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

React:

```
<button onClick={shoot}>Take the Shot!</button>
```

HTML:

```
<button onclick="shoot()">Take the Shot!</button>
```

Event Handlers

A good practice is to put the event handler as a method in the component class:

Example:

Put the `shoot` function inside the `Football` component:

```
class Football extends React.Component {
  shoot() {
    alert("Great Shot!");
  }
  render() {
    return (
      <button onClick={this.shoot}>Take the shot!</button>
    );
  }
}

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

Bind `this`

For methods in React, the `this` keyword should represent the component that owns the method.

That is why you should use arrow functions. With arrow functions, `this` will always represent the object that defined the arrow function.

Example:

```
class Football extends React.Component {
  shoot = () => {
    alert(this);
    /*
    The 'this' keyword refers to the component object
    */
  }
  render() {
    return (
```

```
      <button onClick={this.shoot}>Take the shot!</button>
    );
  }
}

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

Why Arrow Functions?

In class components, the `this` keyword is not defined by default, so with regular functions the `this` keyword represents the object that called the method, which can be the global window object, a HTML button, or whatever.

Read more about binding `this` in our React ES6 'What About this?' chapter.

If you *must* use regular functions instead of arrow functions you have to bind `this` to the component instance using the `bind()` method:

Example:

Make `this` available in the `shoot` function by binding it in the `constructor` function:

```
class Football extends React.Component {
  constructor(props) {
    super(props)
    this.shoot = this.shoot.bind(this)
  }
  shoot() {
    alert(this);
    /*
    Thanks to the binding in the constructor function,
    the 'this' keyword now refers to the component object
    */
  }
  render() {
    return (
      <button onClick={this.shoot}>Take the shot!</button>
    );
  }
```

```
}

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

Without the binding, the `this` keyword would return `undefined`.

Passing Arguments

If you want to send parameters into an event handler, you have two options:

1. Make an anonymous arrow function:

Example:

Send "Goal" as a parameter to the `shoot` function, using arrow function:

```
class Football extends React.Component {
  shoot = (a) => {
    alert(a);
  }
  render() {
    return (
      <button onClick={() => this.shoot("Goal")}>Take the
shot!</button>
    );
  }
}

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

Or:

2. Bind the event handler to `this`.

Note that the first argument has to be `this`.

Example:

Send "Goal" as a parameter to the shoot function:

```
class Football extends React.Component {
  shoot(a) {
    alert(a);
  }
  render() {
    return (
      <button onClick={this.shoot.bind(this, "Goal")}>Take the
shot!</button>
    );
  }
}

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

Note on the second example: If you send arguments without using the `bind` method, the function will be executed at run-time instead of waiting for the button to be clicked.

Sample Program:

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

}
*/
//export default Test;

class DemoApp extends Component {
  constructor(props) {
    super(props);
    this.state = {
      operator: '',
      num1: '',
      num2: '',
      result: '0'
    };
    this.actionHandler = this.actionHandler.bind(this);
    this.handleInputChange = this.handleInputChange.bind(this)
   // this.handleInputChange1 = this.handleInputChange1.bind(this)
  }
```

```
  handleChange =(e)=> {
    this.setState({ operator: e.target.value })
  }

  handleInputChange (e)  {
    this.setState({
      [e.target.name]: Number(e.target.value)
    });

  }

  actionHandler (e) {
    e.preventDefault();


    if(this.state.operator==="+")
    {
    let x = this.state.num1 + this.state.num2;
    this.setState({ result: x })
    }
    else if(this.state.operator==="-")
    {
    let x = this.state.num1 - this.state.num2;
    this.setState({ result: x })
    }
    else if(this.state.operator==="*")
    {
    let x = this.state.num1 * this.state.num2;
    this.setState({ result: x })
    }
    else if(this.state.operator==="/")
    {
    let x = this.state.num1 / this.state.num2;
    this.setState({ result: x })
    }

  }
  render() {
    return (
      <div className="Calculator">
        <form>

          <input type="text" onChange={this.handleInputChange} name="num1"
placeholder="Enter 1st Number" />

          <br />
          <br />
```

```
          <label>
          operator
            <select value={this.state.operator} onChange={this.handleChange} >
              <option value="+" t>+</option>
              <option value="-">-</option>
              <option value="*">*</option>
              <option value="/">/</option>
            </select>
          </label>
          <br />
          <br />
          <input type="text" onChange={this.handleInputChange} name="num2"
placeholder="Enter 2nd Number" />
          <br />
          <br />
          <button onClick={this.actionHandler} type="submit" > Answer
</button>
          <input type="text" value={this.state.result} readOnly />
        </form>
      </div>
    );
  }
}

export default DemoApp;
```