

React Tutorial //

Learning by Examples //

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

```
;"import React from "react" //  
;"import ReactDOM from "react-dom/client" //  
  
} (function Hello(props //  
;<return <h1>Hello World!</h1  //  
{ //  
  
;(("ReactDOM.render(<Hello />, document.getElementById("root //  
  
*/
```

The examples in this tutorial are created using the
.create-react-app

,Install the create-react-app
.and you will be able to run the same examples on your computer

.'In this example we create a component called 'Hello
.'The component is rendered in a container called 'root
/*

Create React App //

.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 tool is an officially supported way to create React applications //

.Node.js is required to use create-react-app //

.Open your terminal in the directory you would like to create your application //

:Run this command to create a React application named my-react-app //

npx create-react-app my-react-app //
.create-react-app will set up everything you need to run a React application //

Note: If you've previously installed create-react-app globally, it is recommended that you //
uninstall the package to ensure npx always uses the latest version of create-react-app. To
.uninstall, run this command: npm uninstall -g create-react-app

Run the React Application //

:Run this command to move to the my-react-app directory //

cd my-react-app //

:Run this command to execute the React application my-react-app //

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

React Introduction //

?What is React //

React, sometimes referred to as a frontend JavaScript framework, 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 V18.0.0 (April 2022 //

.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 //

.(Current version of create-react-app is v5.0.1 (April 2022 //

.create-react-app includes built tools such as webpack, Babel, and ESLint //

React Getting Started //

.To get an overview of what React is, you can write React code directly in HTML //

.But in order to use React in production, you need npm and Node.js installed //

React Directly in HTML //

.The quickest way start learning React is to write React directly in your HTML files //

Start by including three scripts, the first two let us write React code in our JavaScripts, and //
.the third, Babel, allows us to write JSX syntax and ES6 in older browsers

.You will learn more about JSX in the React JSX chapter //

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

<div id="mydiv"></div> //

<"script type="text/babel> //
} ()function Hello //
;return <h1>Hello World!</h1 //
{ //

(('ReactDOM.render(<Hello />, document.getElementById('mydiv //
<script/> //

<body/> //
<html/> //
```

This way of using React can be OK for testing purposes, but for production you will need to //
.set up a React environment

Setting up a React Environment //

If you have npx and Node.js installed, you can create a React application by using //
.create-react-app

If you've previously installed create-react-app globally, it is recommended that you uninstall //
.the package to ensure npx always uses the latest version of create-react-app

.To uninstall, run this command: npm uninstall -g create-react-app //

:Run this command to create a React application named my-react-app //

npx create-react-app my-react-app //

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

Run the React Application //

!Now you are ready to run your first real React application //

:Run this command to move to the my-react-app directory //

cd my-react-app //

:Run this command to run the React application my-react-app //

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

Modify the React Application //

?So far so good, but how do I change the content //

Look in the my-react-app directory, and you will find a src folder. Inside the src folder there //

:is a file called App.js, open it and it will look like this

:myReactApp/src/App.js/ //

```
;'import logo from './logo.svg' //
```

```
;'import './App.css' //
```

```
} ()function App //
```

```
) return //
```

```
<"div className="App" //
```

```
<"header className="App-header" //
```

```
</ "img src={logo} className="App-logo" alt="logo" //
```

```
<p> //
```

```
.Edit <code>src/App.js</code> and save to reload //
```

```
<p/> //
```

```
a> //
```

```
"className="App-link //
```

```
"href="https://reactjs.org //
```

```
"target="_blank //
```

```
"rel="noopener noreferrer //
```

```
< //
```

```
Learn React //
```

```
<a/> //
```

```
<header/> //
```

```
<div/> //
```

```
;( //
```

```
{ //
```

```
;export default App //
```

.Try changing the HTML content and save the file //

Notice that the changes are visible immediately after you save the file, you do not have to //
!reload the browser

.Replace all the content inside the <div className="App"> with a <h1> element //

.See the changes in the browser when you click Save //

```
} ()function App //  
) return //  
<"div className="App"> //  
<h1>Hello World!</h1> //  
<div/> //  
;( //  
{ //  
  
;export default App //
```

.(Notice that we have removed the imports we do not need (logo.svg and App.css //

?What's Next //

Now you have a React Environment on your computer, and you are ready to learn more //
.about React

In the rest of this tutorial we will use our "Show React" tool to explain the various aspects //
.of React, and how they are displayed in the browser

If you want to follow the same steps on your computer, start by stripping down the src //
folder to only contain one file: index.js. You should also remove any unnecessary lines of
code inside the index.js file to make them look like the example in the "Show React" tool
:below

.Click the "Run Example" button to see the result //

:index.js //

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

```
<const myFirstElement = <h1>Hello React!</h1 //
```

```
;(('const root = ReactDOM.createRoot(document.getElementById('root //  
;(root.render(myFirstElement //
```

```
*/
```

You are now watching
'the React file 'index.js

.through our 'Show React' tool
/*

React ES6 //

?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 //

()Array Methods like .map //

Destructuring //

Modules //

Ternary Operator //

Spread Operator //

React ES6 Classes //

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 are assigned inside a constructor() method

:A simple class constructor //

```
} class Car //
```

```
} (constructor(name  //
```

```
;this.brand = name  //
```

```
{  //
```

```
{  //
```

Notice the case of the class name. We have begun the name, "Car", with an uppercase //
.character. This is a standard naming convention for classes

:Now you can create objects using the Car class //

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

```
<DOCTYPE html!> //
```

```
<html> //
```

```

<body> //

<script> //
} class Car //
} (constructor(name //
;this.brand = name //
{ //
{ //

;("const mycar = new Car("Ford //

;(document.write(mycar.brand //
<script/> //

<body/> //
<html/> //

```

.Note: The constructor function is called automatically when the object is initialized //

Method in Classes //

:You can add your own methods in a class //

:"Create a method named "present //

```

<DOCTYPE html!> //
<html> //

<body> //

<script> //
} class Car //
} (constructor(name //
;this.brand = name //
{ //

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

;("const mycar = new Car("Ford //
;(()document.write(mycar.present //
<script/> //

<body/> //
<html/> //

```

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 //

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

```
<DOCTYPE html!> //
<html> //

<body> //

<script> //
} 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 //
{ //
{ //

;("const mycar = new Model("Ford", "Mustang //
;(()document.write(mycar.show //
<script/> //

<body/> //
</html/> //
```

.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 get access to the parent's properties and methods

React ES6 Arrow Functions //

Arrow Functions //

:Arrow functions allow us to write shorter function syntax //

:Before //

```
<DOCTYPE html!> //
```

```
<html> //
```

```
<body> //
```

```
<h1>Function</h1> //
```

```
<p>This demonstrates a regular function, NOT an arrow function.</p> //
```

```
<p id="demo"></p> //
```

```
<script> //
```

```
} ()hello = function //
```

```
;"!return "Hello World" //
```

```
{ //
```

```
;(document.getElementById("demo").innerHTML = hello //
```

```
</script> //
```

```
</body> //
```

```
</html> //
```

:With Arrow Function //

```
<DOCTYPE html!> //
```

```
<html> //
```

```
<body> //
```

```
<h1>Arrow Function</h1> //
```

```
<p>A demonstration of a simple arrow function.</p> //
```

```
<p id="demo"></p> //
```

```
<script> //
```

```
} <= () = hello //
```

```
;"!return "Hello World" //
```

```
{ //
```

```
;(document.getElementById("demo").innerHTML = hello //
```

```
<script/> //
```

```
<body/> //
```

```
<html/> //
```

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 //

```
<DOCTYPE html!> //
```

```
<html> //
```

```
<body> //
```

```
<h1>Arrow Function</h1> //
```

```
<h2>Implicit Return</h2> //
```

p>The arrow function expects a return value, and returns the value by default, without the> //
<return keyword.</p>

```
<p id="demo"></p> //
```

```
<script> //
```

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

```
;)document.getElementById("demo").innerHTML = hello //
```

```
<script/> //
```

```
<body/> //
```

```
<html/> //
```

.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 //

```
<DOCTYPE html!> //
```

```
<html> //
```

```
<body> //
```

```
<h1>Arrow Function</h1> //
```

```
<p>A demonstration of an arrow function in one line, with parameters.</p> //
```

```
<p id="demo"></p> //
```

```
<script> //
```

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

```
;(document.getElementById("demo").innerHTML = hello("World //
```

```
<script/> //
```

```
<body/> //
```

```
<html/> //
```

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

:Arrow Function Without Parentheses //

```
<DOCTYPE html!> //
```

```
<html> //
```

```
<body> //
```

```
<h1>Arrow Function</h1> //
```

p>As you can see in this example, you can skip the parentheses when you have only one> //
<parameter.</p>

```
<p id="demo"></p> //
```

```
<script> //
```

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

```
;(document.getElementById("demo").innerHTML = hello("World //
```

```
<script/> //
```

```
<body/> //
```

```
<html/> //
```

?What About this //

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

.In short, with arrow functions there is 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

:With a regular function, this represents the object that called the function //

```
<DOCTYPE html!> //
```

```
<html> //
```

```
<body> //
```

```
<h1>Regular Function</h1> //
```

p>The this keyword represents different objects depending on how the> //
<function was called.</p>

```
<button id="btn">Click Me!</button> //
```

```
<p><strong>this</strong> represents:</p> //
```

```
<p id="demo"></p> //
```

```
<p>See the difference before and after the button is clicked.</p> //
```

```
<script> //
```

```
} class Header //
```

```
} ()constructor //
```

```
;"this.color = "Red" //
```

```
{ //
```

```
} ()changeColor = function //
```

```
;"document.getElementById("demo").innerHTML += this //
```

```
{ //
```

```
{ //
```

```
;"()const 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 //
```

```
<script/> //
```

```
<body/> //
```

```
<html/> //
```

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

```
<DOCTYPE html!> //
```

```
<html> //
```

```
<body> //
```

```
<h1>Arrow Function</h1> //
```

```
<p>The <strong>this</strong> keyword represents the Header object.</p> //
```

```
<button id="btn">Click Me!</button> //
```

```
<p><strong>this</strong> represents:</p> //
```

```
<p id="demo"></p> //
```

```
<script> //
```

```
} class Header //
```

```
} ()constructor //
```

```
;"this.color = "Red" //
```

```
{ //
```

```
} <= () = changeColor //
```

```
;"document.getElementById("demo").innerHTML += this" //
```

```
{ //
```

```
{ //
```

```
;"()const 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 //
```

```
<script/> //
```

```
<body/> //
```

```
<html/> //
```

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

React ES6 Variables //

Variables //

Before ES6 there was 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 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

.let has a block scope //

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 //

.The keyword const is a bit misleading //

.It does not define a constant value. It defines a constant reference to a value //

:Because of this you can NOT //

Reassign a constant value //

Reassign a constant array //

Reassign a constant object //

:But you CAN //

Change the elements of constant array //

Change the properties of constant object //

React ES6 Array Methods //

Array Methods //

.There are many JavaScript array methods //

.One of the most useful in React is the .map() array method //

The .map() method allows you to run a function on each item in the array, returning a new //
.array as the result

.In React, map() can be used to generate lists //

:Generate a list of items from an array //

```
;'import React from 'react' //
```

```
;'import ReactDOM from 'react-dom/client' //
```

```
;'const myArray = ['apple', 'banana', 'orange' //
```

```
(<const myList = myArray.map((item) => <p>{item}</p> //
```

```
;'ReactDOM.render(myList, document.getElementById('root' //
```

React ES6 Destructuring //

Destructuring //

To illustrate destructuring, we'll make a sandwich. Do you take everything out of the //
refrigerator to make your sandwich? No, you only take out the items you would like to use on
.your sandwich

Destructuring is exactly the same. We may have an array or object that we are working //
.with, but we only need some of the items contained in these

.Destructuring makes it easy to extract only what is needed //

Destructing Arrays //

:Here is the old way of assigning array items to a variable //

```

:Before //

;['const vehicles = ['mustang', 'f-150', 'expedition //

old way //
;[const car = vehicles[0 //
;[const truck = vehicles[1 //
;[const suv = vehicles[2 //
:Here is the new way of assigning array items to a variable //

:With destructuring //

;['const vehicles = ['mustang', 'f-150', 'expedition //

;const [car, truck, suv] = vehicles //
.When destructuring arrays, the order that variables are declared is important //

:If we only want the car and suv we can simply leave out the truck but keep the comma //

;['const vehicles = ['mustang', 'f-150', 'expedition //

;const [car,, suv] = vehicles //
:Destructuring comes in handy when a function returns an array //

<DOCTYPE html!> //
<html> //

<body> //

<script> //
} (function calculate(a, b //
;const add = a + b //
;const subtract = a - b //
;const multiply = a * b //
;const divide = a / b //

;[return [add, subtract, multiply, divide //
{ //

;(const [add, subtract, multiply, divide] = calculate(4, 7 //

;("<document.write("<p>Sum: " + add + "</p //
;("<document.write("<p>Difference " + subtract + "</p //
;("<document.write("<p>Product: " + multiply + "</p //
;("<document.write("<p>Quotient " + divide + "</p //
</script> //

```



```
</body> //  
</html> //
```

Destructuring Objects //

:Here is the old way of using an object inside a function //

:Before //

```
} = const vehicleOne //  
, 'brand: 'Ford' //  
, 'model: 'Mustang' //  
, 'type: 'car' //  
, 'year: 2021' //  
, 'color: 'red' //  
{ //
```

```
;(myVehicle(vehicleOne //
```

old way // //

```
} (function myVehicle(vehicle //  
const message = 'My ' + vehicle.type + ' is a ' + vehicle.color + ' ' + vehicle.brand + ' ' + //  
;' + vehicle.model //  
{ //
```

:Here is the new way of using an object inside a function //

:With destructuring //

```
<!DOCTYPE html> //  
<html> //
```

```
<body> //
```

```
<p id="demo"></p> //
```

```
<script> //  
} = const vehicleOne //  
, 'brand: 'Ford' //  
, 'model: 'Mustang' //  
, 'type: 'car' //  
, 'year: 2021' //  
, 'color: 'red' //  
{ //
```

```
;(myVehicle(vehicleOne //
```

```
} ({function myVehicle({type, color, brand, model //  
;' + const message = 'My ' + type + ' is a ' + color + ' ' + brand + ' ' + model //
```

```

;document.getElementById("demo").innerHTML = message //
{ //
<script/> //

<body/> //
<html/> //

:With destructuring //

<DOCTYPE html!> //
<html> //

<body> //

<p id="demo"></p> //

<script> //
} = const vehicleOne //
,'brand: 'Ford //
,'model: 'Mustang //
,'type: 'car //
,'year: 2021 //
,'color: 'red //
{ //

;(myVehicle(vehicleOne //

} ({function myVehicle({type, color, brand, model //
;.' + const message = 'My ' + type + ' is a ' + color + ' ' + brand + ' ' + model //

;document.getElementById("demo").innerHTML = message //
{ //
<script/> //

<body/> //
<html/> //

```

.Notice that the object properties do not have to be declared in a specific order //

We can even destructure deeply nested objects by referencing the nested object then //
:using a colon and curly braces to again destructure the items needed from the nested object

```

<DOCTYPE html!> //
<html> //

<body> //

```

```

<p id="demo"></p> //

<script> //
} = const vehicleOne //
,'brand: 'Ford //
,'model: 'Mustang //
,'type: 'car //
,'year: 2021 //
,'color: 'red //
} :registration //
,'city: 'Houston //
,'state: 'Texas //
,'country: 'USA //
{ //
{ //

(myVehicle(vehicleOne //

} ({ { function myVehicle({ model, registration: { state //
;',' + const message = 'My ' + model + ' is registered in ' + state //

;document.getElementById("demo").innerHTML = message //
{ //
<script/> //

<body/> //
<html/> //

```

React ES6 Spread Operator //

Spread Operator //

The JavaScript spread operator (...) allows us to quickly copy all or part of an existing array //
 .or object into another array or object

```

<DOCTYPE html!> //
<html> //

<body> //

<script> //
;[const numbersOne = [1, 2, 3 //
;[const numbersTwo = [4, 5, 6 //
;[const numbersCombined = [...numbersOne, ...numbersTwo //

;(document.write(numbersCombined //
<script/> //

<body/> //

```

```
</html/> //
```

.The spread operator is often used in combination with destructuring //

:Assign the first and second items from numbers to variables and put the rest in an array //

```
<DOCTYPE html!> //
```

```
<html> //
```

```
<body> //
```

```
<script> //
```

```
:[const numbers = [1, 2, 3, 4, 5, 6 //
```

```
;const [one, two, ...rest] = numbers //
```

```
;"<document.write("<p>" + one + "</p> //
```

```
;"<document.write("<p>" + two + "</p> //
```

```
;"<document.write("<p>" + rest + "</p> //
```

```
<script/> //
```

```
<body/> //
```

```
</html/> //
```

:We can use the spread operator with objects too //

:Combine these two objects //

```
<DOCTYPE html!> //
```

```
<html> //
```

```
<body> //
```

```
<script> //
```

```
} = const myVehicle //
```

```
, 'brand: 'Ford' //
```

```
, 'model: 'Mustang' //
```

```
'color: 'red' //
```

```
{ //
```

```
} = const updateMyVehicle //
```

```
, 'type: 'car' //
```

```
, 'year: 2021' //
```

```
'color: 'yellow' //
```

```
{ //
```

```
{const myUpdatedVehicle = {...myVehicle, ...updateMyVehicle //
```

```
:Check the result object in the console//  
;(console.log(myUpdatedVehicle //  
<script/> //
```

<p>Press F12 and see the result object in the console view.</p> //

```
<body/> //  
<html/> //
```

Notice the properties that did not match were combined, but the property that did match, //
color, was overwritten by the last object that was passed, updateMyVehicle. The resulting
.color is now yellow

React ES6 Modules //

Modules //

.JavaScript modules allow you to break up your code into separate files //

.This makes it easier to maintain the code-base //

.ES Modules rely on the import and export statements //

Export //

.You can export a function or variable from any file //

.Let us create a file named person.js, and fill it with the things we want to export //

.There are two types of exports: Named and Default //

Named Exports //

.You can create named exports two ways. In-line individually, or all at once at the bottom //

:In-line individually //

person.js //

```
"export const name = "Jesse //
```

```
export const age = 40 //
```

:All at once at the bottom //

person.js //

```
"const name = "Jesse //
```

```
const age = 40 //
```

```
{ export { name, age //
```

Default Exports //

.Let us create another file, named message.js, and use it for demonstrating default export //

.You can only have one default export in a file //

message.js //

```
} <= () = const message //  
;"const name = "Jesse" //  
;const age = 40 //  
;.return name + ' is ' + age + 'years old' //  
;{ //
```

```
;export default message //
```

Import //

You can import modules into a file in two ways, based on if they are named exports or //
.default exports

.Named exports must be destructured using curly braces. Default exports do not //

:Import named exports from the file person.js //

```
<DOCTYPE html!> //
```

```
<html> //
```

```
<body> //
```

```
<p id="demo"></p> //
```

```
<"script type="module> //
```

```
;"import { name, age } from './person.js' //
```

```
;document.getElementById("demo").innerHTML = "My name is " + name //
```

```
;"." + document.getElementById("demo").innerHTML += ", I am " + age //
```

```
<script/> //
```

```
<body/> //
```

```
<html/> //
```

:Import a default export from the file message.js //

```
<DOCTYPE html!> //
```

```
<html> //
```

```
<body> //
```

```
<p id="demo"></p> //
```

```
<"script type="module> //
```

```
;"import message from "./message.js" //

;(document.getElementById("demo").innerHTML = message //

<script/> //

<body/> //
<html/> //
```

React ES6 Ternary Operator //

Ternary Operator //

.The ternary operator is a simplified conditional operator like if / else //

<Syntax: condition ? <expression if true> : <expression if false //

:Here is an example using if / else //

:Before //

```
<DOCTYPE html!> //
<html> //

<body> //

<h1 id="demo"></h1> //

<script> //
} ()function renderApp //
;"!document.getElementById("demo").innerHTML = "Welcome  //
{ //

} ()function renderLogin //
;"document.getElementById("demo").innerHTML = "Please log in  //
{ //

;let authenticated = true //

} (if (authenticated //
;()renderApp  //
} else { //
;()renderLogin  //
{ //

<script/> //
```

p>Try changing the "authenticated" variable to false, and run the code to see what> //
<happens.</p>

```
<body/> //
<html/> //
```

:Here is the same example using a ternary operator //

With Ternary //

```
<DOCTYPE html!> //
<html> //

<body> //

<h1 id="demo"></h1> //

<script> //
} ()function renderApp //
;"!document.getElementById("demo").innerHTML = "Welcome  //
{ //

} ()function renderLogin //
;"document.getElementById("demo").innerHTML = "Please log in  //
{ //

;let authenticated = true //

;(authenticated ? renderApp() : renderLogin //

<script/> //

p>Try changing the "authenticated" variable to false, and run the code to see what> //
<happens.</p>
<body/> //
<html/> //
```

React Render HTML //

.React's goal is in many ways to render HTML in a web page //

.()React renders HTML to the web page by using a function called ReactDOM.render //

The Render Function //

The ReactDOM.render() function takes two arguments, HTML code and an HTML //
.element

The purpose of the function is to display the specified HTML code inside the specified //
.HTML element

?But render where //

There is another folder in the root directory of your React project, named "public". In this //
.folder, there is an index.html file

You'll notice a single <div> in the body of this file. This is where our React application will //
.be rendered

:"Display a paragraph inside an element with the id of "root" //

:"Display a paragraph inside an element with the id of "root" //

```
;(ReactDOM.render(<p>Hello</p>, document.getElementById('root' //  
:The result is displayed in the <div id="root"> element //
```

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

```
;(ReactDOM.render(<p>Hello</p>, document.getElementById('root' //
```

Note that the element id does not have to be called "root", but this is the standard //
.convention

The HTML Code //

The HTML code in this tutorial uses JSX which allows you to write HTML tags inside the //
:JavaScript code

.Do not worry if the syntax is unfamiliar, you will learn more about JSX in the next chapter //

:"Create a variable that contains HTML code and display it in the "root" node //

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

```
) = const myelement //
```

```
<table> //  
<tr> //  
<th>Name</th> //  
<tr> //  
<tr> //  
<td>John</td> //  
<tr> //  
<tr> //  
<td>Elsa</td> //  
<tr> //  
<table/> //  
;( //
```

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

The Root Node //

.The root node is the HTML element where you want to display the result //

.It is like a container for content managed by React //

:'It does NOT have to be a <div> element and it does NOT have to have the id='root //

:The root node can be called whatever you like //

```
<body> */}
```

```
<header id="sandy"></header>
```

```
<body/>
```

```
/* :Display the result in the <header id="sandy"> element
```

```
;'import React from 'react //
```

```
;'import ReactDOM from 'react-dom/client //
```

```
;(ReactDOM.render(<p>Hallo</p>, document.getElementById('sandy //
```

```
*/
```

,For this example to work on your project
you must have a element with

.id="sandy" on your "index.html" page

```
/*
```

React JSX //

?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 //

:Here are two examples. The first uses JSX and the second does not //

```
;'import React from 'react //
```

```

;import ReactDOM from 'react-dom/client' //

;<const myElement = <h1>I Love JSX!</h1 //

;(()=>const root = ReactDOM.createRoot(document.getElementById('root //
;(root.render(myElement //

:JSX //

```

```

;import React from 'react' //
;import ReactDOM from 'react-dom/client' //

;<const myElement = <h1>I Love JSX!</h1 //

;(()=>const root = ReactDOM.createRoot(document.getElementById('root //
;(root.render(myElement //

:Without JSX //

```

```

;import React from 'react' //
;import ReactDOM from 'react-dom/client' //

;(!const myElement = React.createElement('h1', {}, 'I do not use JSX //

;(()=>const root = ReactDOM.createRoot(document.getElementById('root //
;(root.render(myElement //

```

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

:Execute the expression 5 + 5 //

```

;import React from 'react' //
;import ReactDOM from 'react-dom/client' //

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

;(()=>const root = ReactDOM.createRoot(document.getElementById('root //
;(root.render(myElement //

```

Inserting a Large Block of HTML //

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

:Create a list with three list items //

```
;'import React from 'react //
```

```
;'import ReactDOM from 'react-dom/client //
```

```
) = const myElement //
```

```
<ul> //
```

```
<li>Apples</li> //
```

```
<li>Bananas</li> //
```

```
<li>Cherries</li> //
```

```
<ul/> //
```

```
;( //
```

```
;(('const root = ReactDOM.createRoot(document.getElementById('root //
```

```
;(root.render(myElement //
```

One Top Level Element //

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

So if you like to write two paragraphs, you must put them inside a parent element, like a //
.div element

:Wrap two paragraphs inside one DIV element //

```
;'import React from 'react //
```

```
;'import ReactDOM from 'react-do/client //
```

```
) = const myElement //
```

```
<div> //
```

```
<h1>I am a Header.</h1> //
```

```
<h1>I am a Header too.</h1> //
```

```
<div/> //
```

```
;( //
```

```
;(('const root = ReactDOM.createRoot(document.getElementById('root //
```

```
;(root.render(myElement //
```

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

Alternatively, you can use a "fragment" to wrap multiple lines. This will prevent //
.unnecessarily adding extra nodes to the DOM

.</><> :A fragment looks like an empty HTML tag //

:Wrap two paragraphs inside a fragment //

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

```
) = const myElement //  
<> //  
<p>I am a paragraph.</p> //  
<p>I am a paragraph too.</p> //  
</> //  
;( //
```

```
;(('const root = ReactDOM.createRoot(document.getElementById('root' //  
;(root.render(myElement //
```

Elements Must be Closed //

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

</ Close empty elements with //

Result Size: 304 x 530 //

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

```
;</ "const myElement = <input type="text" //
```

```
;(('const root = ReactDOM.createRoot(document.getElementById('root' //  
;(root.render(myElement //
```

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

Attribute class = className //

The class attribute is a much used attribute in HTML, but since JSX is rendered as //
JavaScript, and the class keyword is a reserved word in JavaScript, you are not allowed to
.use it in JSX

.Use attribute className instead //

JSX solved this by using className instead. When JSX is rendered, it translates //
.className attributes into class attributes

:Use attribute className instead of class in JSX //

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

```
;<const myElement = <h1 className="myclass">Hello World</h1 //
```

```
;('const root = ReactDOM.createRoot(document.getElementById('root //  
;(root.render(myElement //
```

Conditions - if statements //

.React supports if statements, but not inside JSX //

To be able to use conditional statements in JSX, you should put the if statements outside //
:of the JSX, or you could use a ternary expression instead

:Option 1 //

:Write if statements outside of the JSX code //

:"Write "Hello" if x is less than 10, otherwise "Goodbye" //

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

```
;const x = 5 //  
;"let text = "Goodbye" //  
} (if (x < 10 //  
;"text = "Hello" //  
{ //
```

```
<const myElement = <h1>{text}</h1 //
```

```
;('const root = ReactDOM.createRoot(document.getElementById('root //  
;(root.render(myElement //
```

:Use ternary expressions instead //

:"Write "Hello" if x is less than 10, otherwise "Goodbye" //

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

```
;const x = 5 //
```

```
<const myElement = <h1>{(x) < 10 ? "Hello" : "Goodbye"}</h1 //
```

```
;('const root = ReactDOM.createRoot(document.getElementById('root //  
;(root.render(myElement //
```

Note that in order to embed a JavaScript expression inside JSX, the JavaScript must be //
.{} ,wrapped with curly braces

React Components //

.Components are like functions that return HTML elements //

React Components //

Components are independent and reusable bits of code. They serve the same purpose as //
.JavaScript functions, but work in isolation and return HTML

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

In older React code bases, you may find Class components primarily used. It is now //
suggested to use Function components along with Hooks, which were added in React 16.8.
.There is an optional section on Class components for your reference

Create Your First Component //

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

Class Component //

A class component must include the extends 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 //

Create a Class component called Car //

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

Function Component //

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

A Function component also returns HTML, and behaves much the same way as a Class //
component, but Function components can be written using much less code, are easier to
.understand, and will be preferred in this tutorial

Create a Function component called Car //

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

Rendering a Component //

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

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

:Display the Car component in the "root" element //

```
;'import React from 'react //
```

```
;'import ReactDOM from 'react-dom/client //
```

```
} ()function Car //
```

```
;<return <h2>Hi, I am a Car!</h2  //
```

```
{ //
```

```
;(('const root = ReactDOM.createRoot(document.getElementById('root //
```

```
;</ root.render(<Car //
```

Props //

.Components can be passed as props, which stands for properties //

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

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

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

```
;'import React from 'react //
```

```
;'import ReactDOM from 'react-dom/client //
```

```
} (function Car(props //
```

```
;<return <h2>I am a {props.color} Car!</h2  //
```

```
{ //
```

```
;(('const root = ReactDOM.createRoot(document.getElementById('root //
```

```
;</"root.render(<Car color="red //
```

Components in Components //

:We can refer to components inside other components //

:Use the Car component inside the Garage component //

```
;'import React from 'react //
```

```
;'import ReactDOM from 'react-dom/client //
```

```
} ()function Car //
```

```
;<return <h2>I am a Car!</h2  //
```

```
{ //
```

```
} ()function Garage //
```

```
) return //
```

```
<>  //
```



```

<h1>Who lives in my Garage?</h1>      //
</ Car>      //
</>    //
;(    //
{ //

;(const root = ReactDOM.createRoot(document.getElementById('root //
;</ root.render(<Garage //

```

Components in Files //

React is all about re-using code, and it is recommended to split your components into //
 .separate files

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

.Note that the filename must start with an uppercase character //

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

```

} ()function Car //
;<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 //

Now we import the "Car.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/client //
;'import Car from './Car.js //

```

```

;(const root = ReactDOM.createRoot(document.getElementById('root //
;</ root.render(<Car //

```

React Class Components //

Before React 16.8, Class components were the only way to track state and lifecycle on a //
 ."React component. Function components were considered "state-less

With the addition of Hooks, Function components are now almost equivalent to Class //
 components. The differences are so minor that you will probably never need to use a Class
 .component in React

Even though Function components are preferred, there are no current plans on removing //
 .Class components from React

.This section will give you an overview of how to use Class components in React //

.Feel free to skip this section, and use Function Components instead //

React Components //

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

Components come in two types, Class components and Function components, in this //
.chapter you will learn about 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 extends 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 //

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 //

:Display the Car component in the "root" element //

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

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

```
;(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

: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 //

```
;'import React from 'react' //  
'import ReactDOM from 'react-dom/client' //  
  
} class Car extends React.Component //  
{ constructor    //  
{ super        //  
{ this.state = {color: "red"    //  
{    //  
{ render    //  
{ return <h2>I am a {this.state.color} Car!</h2    //  
{    //  
{    //
```

```
;(const root = ReactDOM.createRoot(document.getElementById('root' //  
;</ root.render(<Car //
```

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 //

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

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

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

(('ReactDOM.render(<Car color="red"/>, document.getElementById('root //

import React from 'react' //
import ReactDOM from 'react-dom/client' //

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

(('const root = ReactDOM.createRoot(document.getElementById('root //
;</"root.render(<Car model="Mustang //
```

Components in Components //

:We can refer to components inside other components //

:Use the Car component inside the Garage component //

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

} 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>    //
;(    //
{    //
{    //

;(('const root = ReactDOM.createRoot(document.getElementById('root //
;(</ root.render(<Garage //

```

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

:This is the new file, we named it Car.js //

```

;'import React from 'react //

} 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 //

Now we import the Car.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/client //
;'import Car from './Car.js //

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

```

React Class Component State //

.React Class components have a built-in state object //

.You might have noticed that we used state earlier in the component constructor section //

.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 //

:Specify 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 //

: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

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

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

```
} 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/> //
;( //
{ //
{ //
```

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

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

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

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

```
} 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/> //
;( //
{ //
{ //

;(const root = ReactDOM.createRoot(document.getElementById('root //
;</ root.render(<Car //

```

Always use the `setState()` method to change the state object, it will ensure that the component knows its been updated and calls the `render()` method (and all the other lifecycle methods)

Lifecycle of Components //

Each component in React has a lifecycle which you can monitor and manipulate during its three main phases

.The three phases are: Mounting, Updating, and Unmounting //

Mounting //

.Mounting means putting elements into the DOM //

React has four built-in methods that gets called, in this order, when mounting a component

```

()constructor //
()getDerivedStateFromProps //
()render //
()componentDidMount //

```


The render() method is required and will always be called, the others are optional and will //
.be called if you define them

constructor //

The constructor() method is called before anything else, when the component is initiated, //
.and it is the natural place to set up the initial state and other initial values

The constructor() method is called with the props, as arguments, and you should always //
start by calling the super(props) before anything else, this will initiate the parent's constructor
(method and allows the component to inherit methods from its parent (React.Component

:The constructor method is called, by React, every time you make a component //

```
;'import React from 'react' //
```

```
;'import ReactDOM from 'react-dom/client' //
```

```
} class Header extends React.Component //
```

```
{ constructor(props) //
```

```
;(super(props) //
```

```
;'this.state = {favoritecolor: "red" //
```

```
{ //
```

```
} ()render //
```

```
) return //
```

```
<h1>My Favorite Color is {this.state.favoritecolor}</h1> //
```

```
;( //
```

```
{ //
```

```
{ //
```

```
;(const root = ReactDOM.createRoot(document.getElementById('root' //
```

```
;</ root.render(<Header //
```

getDerivedStateFromProps //

The getDerivedStateFromProps() method is called right before rendering the element(s) in //
.the DOM

.This is the natural place to set the state object based on the initial props //

.It takes state as an argument, and returns an object with changes to the state //

The example below starts with the favorite color being "red", but the //

:getDerivedStateFromProps() method updates the favorite color based on the favcol attribute

:The getDerivedStateFromProps method is called right before the render method //

```
;'import React from 'react' //
```

```
;'import ReactDOM from 'react-dom/client' //
```

```
} class Header extends React.Component //
```

```

} (constructor(props //
;(super(props //
;{"this.state = {favoritecolor: "red //
{ //
} (static getDerivedStateFromProps(props, state //
;{ return {favoritecolor: props.favcol //
{ //
} ()render //
) return //
<h1>My Favorite Color is {this.state.favoritecolor}</h1> //
;( //
{ //
{ //

```

```

;('const root = ReactDOM.createRoot(document.getElementById('root //
;</"root.render(<Header favcol="yellow //

```

render //

The render() method is required, and is the method that actually outputs the HTML to the //
.DOM

:A simple component with a simple render() method //

```

;'import React from 'react //
;'import ReactDOM from 'react-dom/client //

} class Header extends React.Component //
} ()render //
) return //
<h1>This is the content of the Header component</h1> //
;( //
{ //
{ //

```

```

;('const root = ReactDOM.createRoot(document.getElementById('root //
;</ root.render(<Header //

```

componentDidMount //

.The componentDidMount() method is called after the component is rendered //

This is where you run statements that requires that the component is already placed in the //
.DOM

:At first my favorite color is red, but give me a second, and it is yellow instead //

```

;'import React from 'react //
;'import ReactDOM from 'react-dom/client //

```

```

} class Header extends React.Component //
} (constructor(props //
;super(props //
;{"this.state = {favoritecolor: "red" //
{ //
} ()componentDidMount //
} <= ()setTimeout //
({"this.setState({favoritecolor: "yellow" //
(1000 ,{ //
{ //
} ()render //
) return //
<h1>My Favorite Color is {this.state.favoritecolor}</h1> //
;( //
{ //
{ //

;((const root = ReactDOM.createRoot(document.getElementById('root //
;(</ root.render(<Header //

```

Updating //

.The next phase in the lifecycle is when a component is updated //

.A component is updated whenever there is a change in the component's state or props //

React has five built-in methods that gets called, in this order, when a component is //
:updated

```

()getDerivedStateFromProps //
()shouldComponentUpdate //
()render //
()getSnapshotBeforeUpdate //
()componentDidUpdate //

```

The render() method is required and will always be called, the others are optional and will //
.be called if you define them

getDerivedStateFromProps //

Also at updates the getDerivedStateFromProps method is called. This is the first method //
.that is called when a component gets updated

.This is still the natural place to set the state object based on the initial props //

The example below has a button that changes the favorite color to blue, but since the //
getDerivedStateFromProps() method is called, which updates the state with the color from
:the favcol attribute, the favorite color is still rendered as yellow

:If the component gets updated, the getDerivedStateFromProps() method is called //

```

;import React from 'react //
;import ReactDOM from 'react-dom/client //

} class Header extends React.Component //
{ constructor(props //
;super(props //
;{"this.state = {favoritecolor: "red //
{ //
} (static getDerivedStateFromProps(props, state //
;{ return {favoritecolor: props.favcol //
{ //
} <= () = changeColor //
;{"this.setState({favoritecolor: "blue //
{ //
} ()render //
) return //
<div> //
<h1>My Favorite Color is {this.state.favoritecolor}</h1> //
<button type="button" onClick={this.changeColor}>Change color</button> //
</div> //
;( //
{ //
{ //

;('const root = ReactDOM.createRoot(document.getElementById('root //
;</ "root.render(<Header favcol="yellow //

```

```

*/
,This example has a button that changes the favorite color to blue
,but since the getDerivedStateFromProps() method is called
the favorite color is still rendered as yellow
because the method updates the state)
.(with the color from the favcol attribute
/*

```

shouldComponentUpdate //

In the shouldComponentUpdate() method you can return a Boolean value that specifies //

.whether React should continue with the rendering or not

.The default value is true //

The example below shows what happens when the shouldComponentUpdate() method //

:returns false

:Stop the component from rendering at any update //

```

;import React from 'react //
;import ReactDOM from 'react-dom/client //

```

```

} class Header extends React.Component //
{ constructor(props //
;super(props //
;{"this.state = {favoritecolor: "red" //
{ //
} ()shouldComponentUpdate //
;return false //
{ //
} <= () = changeColor //
;{"this.setState({favoritecolor: "blue" //
{ //
} ()render //
) return //
<div> //
<h1>My Favorite Color is {this.state.favoritecolor}</h1> //
<button type="button" onClick={this.changeColor}>Change color</button> //
</div> //
;( //
{ //
{ //

;(const root = ReactDOM.createRoot(document.getElementById('root //
;</ root.render(<Header //

```

Same example as above, but this time the shouldComponentUpdate() method returns true // instead

```

;import React from 'react //
;import ReactDOM from 'react-dom/client //

} class Header extends React.Component //
{ constructor(props //
;super(props //
;{"this.state = {favoritecolor: "red" //
{ //
} ()shouldComponentUpdate //
;return true //
{ //
} <= () = changeColor //
;{"this.setState({favoritecolor: "blue" //
{ //
} ()render //
) return //
<div> //
<h1>My Favorite Color is {this.state.favoritecolor}</h1> //
<button type="button" onClick={this.changeColor}>Change color</button> //
</div> //

```

```
;{  //  
{  //  
{  //
```

```
;((const root = ReactDOM.createRoot(document.getElementById('root' //  
;(</ root.render(<Header //
```

render //

The render() method is of course called when a component gets updated, it has to //
.re-render the HTML to the DOM, with the new changes

:The example below has a button that changes the favorite color to blue //

:Click the button to make a change in the component's state //

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

```
} class Header extends React.Component //  
} (constructor(props  //  
;(super(props  //  
;{"this.state = {favoritecolor: "red"  //  
{  //  
} <= () = changeColor  //  
;{"this.setState({favoritecolor: "blue"  //  
{  //  
} ()render  //  
) return  //  
<div>  //  
<h1>My Favorite Color is {this.state.favoritecolor}</h1>  //  
<button type="button" onClick={this.changeColor}>Change color</button>  //  
</div>  //  
;{  //  
{  //  
{  //
```

```
;((const root = ReactDOM.createRoot(document.getElementById('root' //  
;(</ root.render(<Header //
```

getSnapshotBeforeUpdate //

In the getSnapshotBeforeUpdate() method you have access to the props and state before //
the update, meaning that even after the update, you can check what the values were before
.the update

If the getSnapshotBeforeUpdate() method is present, you should also include the //
.componentDidUpdate() method, otherwise you will get an error

:The example below might seem complicated, but all it does is this //

. "When the component is mounting it is rendered with the favorite color "red //

When the component has been mounted, a timer changes the state, and after one second, //
."the favorite color becomes "yellow

This action triggers the update phase, and since this component has a //
getSnapshotBeforeUpdate() method, this method is executed, and writes a message to the
.empty DIV1 element

Then the componentDidUpdate() method is executed and writes a message in the empty //
:DIV2 element

Use the getSnapshotBeforeUpdate() method to find out what the state object looked like //
:before the update

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

} class Header extends React.Component //
} (constructor(props //
;super(props //
;{"this.state = {favoritecolor: "red" //
{ //
} ()componentDidMount //
} <= ()setTimeout //
({"this.setState({favoritecolor: "yellow" //
(1000 ,{ //
{ //
} (getSnapshotBeforeUpdate(prevProps, prevState //
= document.getElementById("div1").innerHTML //
;Before the update, the favorite was " + prevState.favoritecolor" //
{ //
} ()componentDidUpdate //
= document.getElementById("div2").innerHTML //
;The updated favorite is " + this.state.favoritecolor" //
{ //
} ()render //
) return //
<div> //
<h1>My Favorite Color is {this.state.favoritecolor}</h1> //
<div id="div1"></div> //
<div id="div2"></div> //
<div/> //
;( //
{ //
{ //
```

```
;('const root = ReactDOM.createRoot(document.getElementById('root //
;</ root.render(<Header //
```

componentDidUpdate //

.The componentDidUpdate method is called after the component is updated in the DOM //

:The example below might seem complicated, but all it does is this //

."When the component is mounting it is rendered with the favorite color "red //

When the component has been mounted, a timer changes the state, and the color //

."becomes "yellow

This action triggers the update phase, and since this component has a //
componentDidUpdate method, this method is executed and writes a message in the empty
:DIV element

The componentDidUpdate method is called after the update has been rendered in the //
:DOM

```
;'import React from 'react //
```

```
;'import ReactDOM from 'react-dom/client //
```

```
} class Header extends React.Component //
```

```
} (constructor(props //
```

```
;(super(props //
```

```
;'this.state = {favoritecolor: "red" //
```

```
{ //
```

```
} ()componentDidMount //
```

```
} <= ()setTimeout //
```

```
('this.setState({favoritecolor: "yellow" //
```

```
(1000 ,{ //
```

```
{ //
```

```
} ()componentDidUpdate //
```

```
= document.getElementById("mydiv").innerHTML //
```

```
;'The updated favorite is " + this.state.favoritecolor" //
```

```
{ //
```

```
} ()render //
```

```
) return //
```

```
<div> //
```

```
<h1>My Favorite Color is {this.state.favoritecolor}</h1> //
```

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

```
</div> //
```

```
;' //
```

```
{ //
```

```
{ //
```

```
;('const root = ReactDOM.createRoot(document.getElementById('root //
```



```
;(</ root.render(<Header //
```

Unmounting //

The next phase in the lifecycle is when a component is removed from the DOM, or //
.unmounting as React likes to call it

:React has only one built-in method that gets called when a component is unmounted //

```
()componentWillUnmount //
```

```
componentWillUnmount //
```

The componentWillUnmount method is called when the component is about to be removed //
.from the DOM

:Click the button to delete the header //

```
;'import React from 'react //
```

```
;'import ReactDOM from 'react-dom/client //
```

```
} class Container extends React.Component //
```

```
} (constructor(props //
```

```
;(super(props //
```

```
;(this.state = {show: true //
```

```
{ //
```

```
} <= () = delHeader //
```

```
;(this.setState({show: false //
```

```
{ //
```

```
} ()render //
```

```
;let myheader //
```

```
} (if (this.state.show //
```

```
; </ myheader = <Child //
```

```
;( //
```

```
) return //
```

```
<div> //
```

```
{myheader} //
```

```
<button type="button" onClick={this.delHeader}>Delete Header</button> //
```

```
<div/> //
```

```
;( //
```

```
{ //
```

```
{ //
```

```
} class Child extends React.Component //
```

```
} ()componentWillUnmount //
```

```
;(".alert("The component named Header is about to be unmounted //
```

```
{ //
```

```
} ()render //
```

```
) return //
```

```
<h1>Hello World!</h1> //
```

```
;( //
```

```
{ //
{ //
```

```
;(const root = ReactDOM.createRoot(document.getElementById('root //
;</ root.render(<Container //
```

React Props //

.Props are arguments passed into React components //

.Props are passed to components via HTML attributes //

.props stands for properties //

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 //

:Add a "brand" attribute to the Car element //

```
;</ "const myElement = <Car brand="Ford //
```

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

:Use the brand attribute in the component //

```
;'import React from 'react //
```

```
;'import ReactDOM from 'react-dom/client //
```

```
} (function Car(props //
```

```
;<return <h2>I am a { props.brand }!</h2  //
```

```
{ //
```

```
;</ "const myElement = <Car brand="Ford //
```

```
;(const root = ReactDOM.createRoot(document.getElementById('root //
```

```
;(root.render(myElement //
```

Pass Data //

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

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

```
;'import React from 'react //
```

```
;'import ReactDOM from 'react-dom/client //
```

```
} (function Car(props //
```

```

    <return <h2>I am a { props.brand }!</h2> //
    { //

    } ()function Garage //
    ) return //
    <> //
    <h1>Who lives in my garage?</h1> //
    </ "Car brand="Ford> //
    </> //
    ;{ //
    { //

    ;(const root = ReactDOM.createRoot(document.getElementById('root //
    ;(</ root.render(<Garage //

```

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

:Create a variable named carName and send it to the Car component //

```

;import React from 'react //
;import ReactDOM from 'react-dom/client //

} (function Car(props //
;return <h2>I am a { props.brand }!</h2> //
{ //

} ()function Garage //
;"const carName = "Ford" //
) return //
<> //
<h1>Who lives in my garage?</h1> //
</ { Car brand={ carName}> //
</> //
;{ //
{ //

;(const root = ReactDOM.createRoot(document.getElementById('root //
;(</ root.render(<Garage //

```

:Or if it was an object //

:Create an object named carInfo and send it to the Car component //

```

;import React from 'react //
;import ReactDOM from 'react-dom/client //

} (function Car(props //

```

```

; <return <h2>I am a { props.brand.model }!</h2> //
{ //

} ()function Garage //
; { "const carInfo = { name: "Ford", model: "Mustang" //
} return //
<> //
<h1>Who lives in my garage?</h1> //
</ { Car brand={ carInfo}> //
</> //
; { //
{ //

; (const root = ReactDOM.createRoot(document.getElementById('root' //
; </ root.render(<Garage //

```

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

React Events //

.Just like HTML DOM events, 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> //
```

:Put the shoot function inside the Football component //

```
import React from 'react' //
```

```
import ReactDOM from 'react-dom/client' //
```

```
()function Football //
```

```
<= () = const shoot //
```

```
!alert("Great Shot" //
```

```
{ //
```

```

) return //
<button onClick={shoot}>Take the shot!</button> //
;( //
{ //

;(()=>const root = ReactDOM.createRoot(document.getElementById('root //
;(</ root.render(<Football //

```

Passing Arguments //

.To pass an argument to an event handler, use an arrow function //

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

```

;import React from 'react //
;import ReactDOM from 'react-dom/client //

```

```

} ()function Football //
} <= (const shoot = (a //
;(alert(a //
{ //

```

```

) return //
<button onClick={() => shoot("Goal!")}>Take the shot!</button> //
;( //
{ //

```

```

;(()=>const root = ReactDOM.createRoot(document.getElementById('root //
;(</ root.render(<Football //

```

React Event Object //

.Event handlers have access to the React event that triggered the function //

.In our example the event is the "click" event //

:Arrow Function: Sending the event object manually //

```

;import React from 'react //
;import ReactDOM from 'react-dom/client //

```

```

} ()function Football //
} <= (const shoot = (a, b //
;(alert(b.type //
/* //

```

.b' represents the React event that triggered the function' //

In this case, the 'click' event //

```

/* //
{ //

```

```

) return //
<button onClick={(event) => shoot("Goal!", event)}>Take the shot!</button> //
;( //
{ //

```

```

;('const root = ReactDOM.createRoot(document.getElementById('root //
;</ root.render(<Football //

```

.This will come in handy when we look at Form in a later chapter //

React Conditional Rendering //

.In React, you can conditionally render components //

.There are several ways to do this //

if Statement //

.We can use the if JavaScript operator to decide which component to render //

:We'll use these two components //

```

} ()function MissedGoal //
;<return <h1>MISSED!</h1 //
{ //

```

```

} ()function MadeGoal //
;<return <h1>Goal!</h1 //
{ //

```

Now, we'll create another component that chooses which component to render based on a //
:condition

```

;import React from 'react //
;import ReactDOM from 'react-dom/client //

```

```

} ()function MissedGoal //
;<return <h1>MISSED!</h1 //
{ //

```

```

} ()function MadeGoal //
;<return <h1>GOAL!</h1 //
{ //

```

```

} (function Goal(props //
;const isGoal = props.isGoal //
} if (isGoal //
;</return <MadeGoal //
{ //

```

```

</return <MissedGoal //
{ //

;('const root = ReactDOM.createRoot(document.getElementById('root //
</ {root.render(<Goal isGoal={false //

:Try changing the isGoal attribute to true //

;import React from 'react //
;import ReactDOM from 'react-dom/client //

} ()function MissedGoal //
<return <h1>MISSED!</h1 //
{ //

} ()function MadeGoal //
<return <h1>GOAL!</h1 //
{ //

} (function Goal(props //
;const isGoal = props.isGoal //
} (if (isGoal //
<return <MadeGoal //
{ //
<return <MissedGoal //
{ //

;('const root = ReactDOM.createRoot(document.getElementById('root //
</ {root.render(<Goal isGoal={true //

```

Logical && Operator //

.Another way to conditionally render a React component is by using the && operator //

:We can embed JavaScript expressions in JSX by using curly braces //

```

;import React from 'react //
;import ReactDOM from 'react-dom/client //

} (function Garage(props //
;const cars = props.cars //
) return //
<> //
<h1>Garage</h1> //
&& cars.length > 0} //
<h2> //
.You have {cars.length} cars in your garage //
<h2/> //
{ //

```

```

</> //
;( //
{ //

;['const cars = ['Ford', 'BMW', 'Audi //
;('const root = ReactDOM.createRoot(document.getElementById('root //
;</ {root.render(<Garage cars={cars //

```

.If cars.length > 0 is equates to true, the expression after && will render //

:Try emptying the cars array //

```

;import React from 'react //
;import ReactDOM from 'react-dom/client //

} (function Garage(props //
;const cars = props.cars //
) return //
<> //
<h1>Garage</h1> //
&& cars.length > 0} //
<h2> //
.You have {cars.length} cars in your garage //
<h2/> //
{ //
</> //
;( //
{ //

```

```

;[] = const cars //
;('const root = ReactDOM.createRoot(document.getElementById('root //
;</ {root.render(<Garage cars={cars //

```

Ternary Operator //

.Another way to conditionally render elements is by using a ternary operator //

```

condition ? true : false //
.We will go back to the goal example //

```

Return the MadeGoal component if isGoal is true, otherwise return the MissedGoal //

:component

```

;import React from 'react //
;import ReactDOM from 'react-dom/client //

} ()function MissedGoal //
;<return <h1>MISSED!</h1 //
{ //

```



```

} ()function MadeGoal //
;return <h1>GOAL!</h1> //
{ //

} (function Goal(props //
;const isGoal = props.isGoal //
) return //
<> //
{ </isGoal ? <MadeGoal/> : <MissedGoal > //
</> //
;( //
{ //

;(()const root = ReactDOM.createRoot(document.getElementById('root //
;</> {root.render(<Goal isGoal={false} //

```

React Lists //

.In React, you will render lists with some type of loop //

.The JavaScript map() array method is generally the preferred method //

:Let's render all of the cars from our garage //

```

;import React from 'react //
;import ReactDOM from 'react-dom/client //

} (function Car(props //
;return <li>I am a { props.brand }</li> //
{ //

} ()function Garage //
;[const cars = ['Ford', 'BMW', 'Audi //
) return //
<> //
<h1>Who lives in my garage?</h1> //
<ul> //
{(</> {cars.map((car) => <Car brand={car} //
<ul/> //
</> //
;( //
{ //

;(()const root = ReactDOM.createRoot(document.getElementById('root //
;</> root.render(<Garage //

*/

```

,If you run this example in your create-react-app
.you will receive a warning that there is no "key" provided for the list items
/*

When you run this code in your create-react-app, it will work but you will receive a warning //
.that there is no "key" provided for the list items

Keys //

Keys allow React to keep track of elements. This way, if an item is updated or removed, //
.only that item will be re-rendered instead of the entire list

.Keys need to be unique to each sibling. But they can be duplicated globally //

Generally, the key should be a unique ID assigned to each item. As a last resort, you can //
.use the array index as a key

:Let's refactor our previous example to include keys //

```
;'import React from 'react' //  
;'import ReactDOM from 'react-dom/client' //  
  
{ (function Car(props) //  
;return <li>I am a { props.brand }</li> //  
{ //  
  
} (function Garage //  
] = const cars //  
,{'id: 1, brand: 'Ford'} //  
,{'id: 2, brand: 'BMW'} //  
{'id: 3, brand: 'Audi'} //  
;[ //  
) return //  
<> //  
<h1>Who lives in my garage?</h1> //  
<ul> //  
{(</ {cars.map((car) => <Car key={car.id} brand={car.brand} //  
<ul/> //  
</> //  
;{ //  
{ //
```

React Forms //

.Just like in HTML, React uses forms to allow users to interact with the web page //

Adding Forms in React //

:You add a form with React like any other element //

:Add a form that allows users to enter their name //

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

} ()function MyForm //
) return //
<form> //
:label>Enter your name> //
</ "input type="text> //
<label/> //
<form/> //
( //
{ //

;('const root = ReactDOM.createRoot(document.getElementById('root' //
;(</ root.render(<MyForm //
```

.This will work as normal, the form will submit and the page will refresh //

.But this is generally not what we want to happen in React //

.We want to prevent this default behavior and let React control the form //

Handling Forms //

.Handling forms is about how you handle the data when it changes value or gets submitted //

.In HTML, form data is usually handled by the DOM //

.In React, form data is usually handled by the components //

When the data is handled by the components, all the data is stored in the component //

.state

.You can control changes by adding event handlers in the onChange attribute //

We can use the useState Hook to keep track of each inputs value and provide a "single //

.source of truth" for the entire application

:Use the useState Hook to manage the input //

```
;"import { useState } from "react" //
import ReactDOM from 'react-dom/client' //

} ()function MyForm //
;('')const [name, setName] = useState //

) return //
```

```

<form>    //
:label>Enter your name>    //
input>    //
"type="text    //
{value={name    //
{(onChange={e => setName(e.target.value    //
</    //
<label/>    //
<form/>    //
(    //
{    //

```

```

;(const root = ReactDOM.createRoot(document.getElementById('root //
;</ root.render(<MyForm //

```

Submitting Forms //

You can control the submit action by adding an event handler in the onSubmit attribute for //

```

:<the <form

```

:Add a submit button and an event handler in the onSubmit attribute //

```

;"import { useState } from "react //
;import ReactDOM from 'react-dom/client //

} ()function MyForm //
;(")const [name, setName] = useState //

} <= (const handleSubmit = (event //
;()event.preventDefault //
;({alert(`The name you entered was: ${name //
{ //

) return //
<{form onSubmit={handleSubmit} //
:label>Enter your name> //
input> //
"type="text //
{value={name //
{(onChange={e => setName(e.target.value //
</ //
<label/> //
</ "input type="submit> //
<form/> //
( //
{ //

```

Multiple Input Fields //

You can control the values of more than one input field by adding a name attribute to each `.element`

.We will initialize our state with an empty object //

To access the fields in the event handler use the `event.target.name` and `event.target.value` //
.syntax

.To update the state, use square brackets [bracket notation] around the property name //

:Write a form with two input fields //

```
;"import { useState } from "react" //
;"import ReactDOM from "react-dom/client" //

} ()function MyForm //
;({})const [inputs, setInputs] = useState //

} <= (const handleChange = (event //
;const name = event.target.name //
;const value = event.target.value //
(({setInputs(values => ({...values, [name]: value //
{ //

} <= (const handleSubmit = (event //
;()event.preventDefault //
;(console.log(inputs //
{ //

) return //
<{form onSubmit={handleSubmit} //
:label>Enter your name> //
input> //
"type="text //
"name="username //
{"" || value={inputs.username //
{onChange={handleChange //
</ //
<label/> //
:label>Enter your age> //
input> //
"type="number //
"name="age //
{"" || value={inputs.age //
{onChange={handleChange //
</ //
<label/> //
</ "input type="submit"> //
```

```

</form> //
( //
{ //

;(()=>const root = ReactDOM.createRoot(document.getElementById('root //
;</ root.render(<MyForm //

*/
"Click F12 and navigate to the "Console view
.to see the result when you submit the form
/*

```

Note: We use the same event handler function for both input fields, we could write one // event handler for each, but this gives us much cleaner code and is the preferred way in .React

Textarea //
 .The textarea element in React is slightly different from ordinary HTML //

In HTML the value of a textarea was the text between the start tag <textarea> and the end // .</tag </textarea>

```

<textarea> //
.Content of the textarea //
</textarea> //

```

In React the value of a textarea is placed in a value attribute. We'll use the useState Hook // :to manage the value of the textarea

:A simple textarea with some content //

```

;"import { useState } from "react //
;"import ReactDOM from "react-dom/client //

```

```

} ()function MyForm //
)const [textarea, setTextarea] = useState //
"The content of a textarea goes in the value attribute" //
;( //

```

```

} <= (const handleChange = (event //
(setTextarea(event.target.value //
{ //

```

```

) return //
</form> //
</ {textarea value={textarea} onChange={handleChange} //
</form/> //
( //
{ //

```

```
;('const root = ReactDOM.createRoot(document.getElementById('root //
;(</ root.render(<MyForm //
```

Select //

.A drop down list, or a select box, in React is also a bit different from HTML //

:in HTML, the selected value in the drop down list was defined with the selected attribute //

```
:HTML //
<select> //
<option value="Ford">Ford</option> //
<option value="Volvo" selected>Volvo</option> //
<option value="Fiat">Fiat</option> //
<select/> //
```

:In React, the selected value is defined with a value attribute on the select tag //

:A simple select box, where the selected value "Volvo" is initialized in the constructor //

```
;"import { useState } from "react //
;"import ReactDOM from "react-dom/client //
```

```
} ()function MyForm //
;"const [myCar, setMyCar] = useState("Volvo //

} <= (const handleChange = (event //
(setMyCar(event.target.value //
{ //

) return //
<form> //
<{select value={myCar} onChange={handleChange> //
<option value="Ford">Ford</option> //
<option value="Volvo">Volvo</option> //
<option value="Fiat">Fiat</option> //
<select/> //
<form/> //
( //
{ //
```

```
;('const root = ReactDOM.createRoot(document.getElementById('root //
;(</ root.render(<MyForm //
```

By making these slight changes to <textarea> and <select>, React is able to handle all //
.input elements in the same way

React Router //

.Create React App doesn't include page routing //

.React Router is the most popular solution //

Add React Router //

To add React Router in your application, run this in the terminal from the root directory of //
:the application

```
npm i -D react-router-dom //
```

.Note: This tutorial uses React Router v6 //

:If you are upgrading from v5, you will need to use the @latest flag //

```
npm i -D react-router-dom@latest //
```

Folder Structure //

.To create an application with multiple page routes, let's first start with the file structure //

:Within the src folder, we'll create a folder named pages with several files //

:\src\pages //

Layout.js //

Home.js //

Blogs.js //

Contact.js //

NoPage.js //

.Each file will contain a very basic React component //

Basic Usage //

.Now we will use our Router in our index.js file //

:Use React Router to route to pages based on URL //

:index.js //

```
;"import ReactDOM from "react-dom/client" //
```

```
;"import { BrowserRouter, Routes, Route } from "react-router-dom" //
```

```
;"import Layout from "../pages/Layout" //
```

```
;"import Home from "../pages/Home" //
```

```
;"import Blogs from "../pages/Blogs" //
```

```
;"import Contact from "../pages/Contact" //
```

```
;"import NoPage from "../pages/NoPage" //
```

```
} ()export default function App //
```

```
) return //
```

```
<BrowserRouter> //
```

```
<Routes> //
```



```

<{</ Route path="/" element={<Layout>      //
</ {</ Route index element={<Home>          //
</ {</ Route path="blogs" element={<Blogs>    //
</ {</ Route path="contact" element={<Contact> //
</ {</ Route path="*" element={<NoPage>      //
<Route/>      //
<Routes/>      //
<BrowserRouter/>  //
;(  //
{  //

;(const root = ReactDOM.createRoot(document.getElementById('root //
;(root.render(<App //

```

Example Explained //

.<We wrap our content first with <BrowserRouter //

Then we define our <Routes>. An application can have multiple <Routes>. Our basic //
 .example only uses one

Route>s can be nested. The first <Route> has a path of / and renders the Layout> //
 .component

The nested <Route>s inherit and add to the parent route. So the blogs path is combined //
 .with the parent and becomes /blogs

The Home component route does not have a path but has an index attribute. That //
 ./ specifies this route as the default route for the parent route, which is

Setting the path to * will act as a catch-all for any undefined URLs. This is great for a 404 //
 .error page

Pages / Components //

.The Layout component has <Outlet> and <Link> elements //

.The <Outlet> renders the current route selected //

.<Link> is used to set the URL and keep track of browsing history> //

.<""=Anytime we link to an internal path, we will use <Link> instead of <a href //

The "layout route" is a shared component that inserts common content on all pages, such //
 .as a navigation menu

:Layout.js //

;"import { Outlet, Link } from "react-router-dom //

```

} <= () = const Layout //
) return //
<> //
<nav> //
<ul> //
<li> //
<Link to="/">Home</Link> //
</li> //
<li> //
<Link to="/blogs">Blogs</Link> //
</li> //
<li> //
<Link to="/contact">Contact</Link> //
</li> //
<ul/> //
<nav/> //

</ Outlet> //
</> //
( //
;{ //

;export default Layout //
:Home.js //

} <= () = const Home //
;<return <h1>Home</h1> //
;{ //

;export default Home //
:Blogs.js //

} <= () = const Blogs //
;<return <h1>Blog Articles</h1> //
;{ //

;export default Blogs //
:Contact.js //

} <= () = const Contact //
;<return <h1>Contact Me</h1> //
;{ //

;export default Contact //
:NoPage.js //

} <= () = const NoPage //
;<return <h1>404</h1> //

```

```

;{ //

;export default NoPage //

React Memo //

.Using memo will cause React to skip rendering a component if its props have not changed //

.This can improve performance //

This section uses React Hooks. See the React Hooks section for more information on //
.Hooks

Problem //
.In this example, the Todos component re-renders even when the todos have not changed //

:Example //
:index.js //

;"import { useState } from "react //
;"import ReactDOM from "react-dom/client //
;"import Todos from "../Todos //

} <= () = const App //
;(const [count, setCount] = useState(0 //
;(["const [todos, setTodos] = useState(["todo 1", "todo 2 //

} <= () = const increment //
;(setCount((c) => c + 1 //
;{ //

) return //
<> //
</ {Todos todos={todos} //
</ hr> //
<div> //
{Count: {count //
<button onClick={increment}>+</button> //
<div/> //
</> //
;( //
;{ //

;((const root = ReactDOM.createRoot(document.getElementById('root //
;(</ root.render(<App //
:Todos.js //

;"import { useState } from "react //

```

```

;import ReactDOM from "react-dom/client" //
;import Todos from "./Todos" //

} <= () = const App //
;(const [count, setCount] = useState(0 //
;(["const [todos, setTodos] = useState(["todo 1", "todo 2 //

} <= () = const increment //
;(setCount((c) => c + 1 //
;{ //

) return //
<> //
</ {Todos todos={todos} //
</ hr> //
<div> //
{Count: {count //
<button onClick={increment}>+</button> //
<div/> //
</> //
;{ //
;{ //

;((const root = ReactDOM.createRoot(document.getElementById('root //
;(</ root.render(<App //

```

.When you click the increment button, the Todos component re-renders //

.If this component was complex, it could cause performance issues //

Solution //

.To fix this, we can use memo //

.Use memoto keep the Todos component from needlessly re-rendering //

:Wrap the Todos component export in memo //

:index.js //

```

;import { useState } from "react" //
;import ReactDOM from "react-dom/client" //
;import Todos from "./Todos" //

} <= () = const App //
;(const [count, setCount] = useState(0 //
;(["const [todos, setTodos] = useState(["todo 1", "todo 2 //

} <= () = const increment //

```

```

;(setCount((c) => c + 1    //
;{    //

) return    //
<>    //
</ {Todos todos={todos}>    //
</ hr>    //
<div>    //
{Count: {count    //
<button onClick={increment}>+</button>    //
<div/>    //
</>    //
;(    //
;{    //

;('const root = ReactDOM.createRoot(document.getElementById('root //
;(</ root.render(<App //
:Todos.js //

;"import { useState } from "react //
;"import ReactDOM from "react-dom/client //
;"import Todos from "./Todos //

} <= () = const App //
;(const [count, setCount] = useState(0    //
;(["const [todos, setTodos] = useState(["todo 1", "todo 2    //

} <= () = const increment    //
;(setCount((c) => c + 1    //
;{    //

) return    //
<>    //
</ {Todos todos={todos}>    //
</ hr>    //
<div>    //
{Count: {count    //
<button onClick={increment}>+</button>    //
<div/>    //
</>    //
;(    //
;{    //

;('const root = ReactDOM.createRoot(document.getElementById('root //
;(</ root.render(<App //

```

Now the Todos component only re-renders when the todos that are passed to it through `.props` are updated

Styling React Using CSS //

There are many ways to style React with CSS, this tutorial will take a closer look at three //
:common ways

Inline styling //

CSS stylesheets //

CSS Modules //

Inline Styling //

:To style an element with the inline style attribute, the value must be a JavaScript object //

:Insert an object with the styling information //

```
;'import React from 'react //
```

```
;'import ReactDOM from 'react-dom/client //
```

```
} <= () = const Header //
```

```
) return //
```

```
<> //
```

```
<h1 style={{color: "red"}}>Hello Style!</h1> //
```

```
<p>Add a little style!</p> //
```

```
</> //
```

```
;( //
```

```
{ //
```

```
;(const root = ReactDOM.createRoot(document.getElementById('root //
```

```
);(</ root.render(<Header //
```

Note: In JSX, JavaScript expressions are written inside curly braces, and since JavaScript //
objects also use curly braces, the styling in the example above is written inside two sets of
.{{}} curly braces

camelCased Property Names //

Since the inline CSS is written in a JavaScript object, properties with hyphen separators, //

:like background-color, must be written with camel case syntax

:Use backgroundColor instead of background-color //

```
;'import React from 'react //
```

```
;'import ReactDOM from 'react-dom/client //
```

```
} <= () = const Header //
```

```
) return //
```

```
<> //
```

```
<h1 style={{backgroundColor: "lightblue"}}>Hello Style!</h1> //
```

```
<p>Add a little style!</p> //
```

```
</> //
```

```
;{ //
{ //
```

```
;('const root = ReactDOM.createRoot(document.getElementById('root //
;(</ root.render(<Header //
```

JavaScript Object //

:You can also create an object with styling information, and refer to it in the style attribute //

:Create a style object named myStyle //

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

```
} <= () = const Header //
} = const myStyle //
,"color: "white //
,"backgroundColor: "DodgerBlue //
,"padding: "10px //
"fontFamily: "Sans-Serif //
;{ //
) return //
<> //
<h1 style={myStyle}>Hello Style!</h1> //
<p>Add a little style!</p> //
</> //
;{ //
{ //
```

```
;('const root = ReactDOM.createRoot(document.getElementById('root //
;(</ root.render(<Header //
```

CSS Stylesheet //

You can write your CSS styling in a separate file, just save the file with the .css file //
.extension, and import it in your application

:App.css //

:Create a new file called "App.css" and insert some CSS code in it //

```
} body //
;background-color: #282c34 //
;color: white //
;padding: 40px //
;font-family: Sans-Serif //
;text-align: center //
{ //
```

.Note: You can call the file whatever you like, just remember the correct file extension //

:Import the stylesheet in your application //

```
;'import React from 'react' //
;'import ReactDOM from 'react-dom/client' //
;'import './App.css' //
```

```
} <= () = const Header //
) return //
<> //
<h1>Hello Style!</h1> //
<p>Add a little style!</p> //
</> //
;( //
{ //
```

```
;(const root = ReactDOM.createRoot(document.getElementById('root' //
);</ root.render(<Header //
```

CSS Modules //

.Another way of adding styles to your application is to use CSS Modules //

.CSS Modules are convenient for components that are placed in separate files //

The CSS inside a module is available only for the component that imported it, and you do //
.not have to worry about name conflicts

.Create the CSS module with the .module.css extension, example: my-style.module.css //

:Create a new file called "my-style.module.css" and insert some CSS code in it //

```
:my-style.module.css //
} bigblue. //
;color: DodgerBlue //
;padding: 40px //
;font-family: Sans-Serif //
;text-align: center //
{ //
```

:Import the stylesheet in your component //

```
:Car.js //
;'import styles from './my-style.module.css' //
```

```
} <= () = const Car //
;<return <h1 className={styles.bigblue}>Hello Car!</h1 //
{ //
```



```
;export default Car //
```

```
:Import the component in your application //
```

```
:index.js //
```

```
;'import React from 'react' //
```

```
;'import ReactDOM from 'react-dom/client' //
```

```
;'import Car from './Car.js' //
```

```
;(function()const root = ReactDOM.createRoot(document.getElementById('root' //
```

```
);(</ root.render(<Car //
```

Styling React Using Sass //

What is Sass //

.Sass is a CSS pre-processor //

.Sass files are executed on the server and sends CSS to the browser //

.You can learn more about Sass in our Sass Tutorial //

?Can I use Sass //

If you use the create-react-app in your project, you can easily install and use Sass in your //

.React projects

:Install Sass by running this command in your terminal //

```
npm i sass< //
```

!Now you are ready to include Sass files in your project //

Create a Sass file //

Create a Sass file the same way as you create CSS files, but Sass files have the file //
extension .scss

:In Sass files you can use variables and other Sass functions //

Example //

```
:my-sass.scss //
```

:Create a variable to define the color of the text //

```
;myColor: red$ //
```

```
} h1 //
```

```
;color: $myColor //
```

```
{ //
```

:Import the Sass file the same way as you imported a CSS file //

```
;'import React from 'react' //
;'import ReactDOM from 'react-dom/client' //
;'import './my-sass.scss' //
```

```
} <= () = const Header //
) return //
<> //
<h1>Hello Style!</h1> //
<p>Add a little style!</p> //
</> //
;( //
{ //
```

```
;(const root = ReactDOM.createRoot(document.getElementById('root' //
;</ root.render(<Header //
```

React Hooks //

.Hooks were added to React in version 16.8 //

Hooks allow function components to have access to state and other React features. //
.Because of this, class components are generally no longer needed

Although Hooks generally replace class components, there are no plans to remove classes //
.from React

?What is a Hook //

.Hooks allow us to "hook" into React features such as state and lifecycle methods //

Here is an example of a Hook. Don't worry if it doesn't make sense. We will go into more //
.detail in the next section

```
;"import React, { useState } from 'react' //
;"import ReactDOM from 'react-dom/client' //
```

```
} ()function FavoriteColor //
;('const [color, setColor] = useState("red" //

) return //
<> //
<h1>My favorite color is {color}!</h1> //
button> //
"type="button //
{("onClick={() => setColor("blue //
<Blue</button< //
button> //
```

```

"type="button      //
{"onClick={() => setColor("red      //
<Red</button<      //
button>      //
"type="button      //
{"onClick={() => setColor("pink      //
<Pink</button<      //
button>      //
"type="button      //
{"onClick={() => setColor("green      //
<Green</button<      //
</>      //
;{      //
{      //

;('const root = ReactDOM.createRoot(document.getElementById('root //
;</ root.render(<FavoriteColor //

```

.You must import Hooks from react //

.Here we are using the useState Hook to keep track of the application state //

.State generally refers to application data or properties that need to be tracked //

Hook Rules //

:There are 3 rules for hooks //

.Hooks can only be called inside React function components //

.Hooks can only be called at the top level of a component //

Hooks cannot be conditional //

.Note: Hooks will not work in React class components //

Custom Hooks //

If you have stateful logic that needs to be reused in several components, you can build //

.your own custom Hooks

React useState Hook //

.The React useState Hook allows us to track state in a function component //

.State generally refers to data or properties that need to be tracking in an application //

Import useState //

.To use the useState Hook, we first need to import it into our component //

.At the top of your component, import the useState Hook //

```
;"import { useState } from "react //
```

.Notice that we are destructuring useState from react as it is a named export //

.To learn more about destructuring, check out the ES6 section //

Initialize useState //

.We initialize our state by calling useState in our function component //

:useState accepts an initial state and returns two values //

.The current state //

.A function that updates the state //

.Initialize state at the top of the function component //

```
;"import { useState } from "react //
```

```
} ()function FavoriteColor //
```

```
;"const [color, setColor] = useState //
```

```
{ //
```

.Notice that again, we are destructuring the returned values from useState //

.The first value, color, is our current state //

.The second value, setColor, is the function that is used to update our state //

.These names are variables that can be named anything you would like //

("")Lastly, we set the initial state to an empty string: useState //

Read State //

.We can now include our state anywhere in our component //

.Use the state variable in the rendered component //

```
;"import { useState } from "react //
```

```
;"import ReactDOM from "react-dom/client //
```

```
} ()function FavoriteColor //
```

```
;"const [color, setColor] = useState("red //
```

```
<return <h1>My favorite color is {color}!</h1 //
```

```
{ //
```

```
;(const root = ReactDOM.createRoot(document.getElementById('root //
```

```
;</ root.render(<FavoriteColor //
```

Update State //

.To update our state, we use our state updater function //

.We should never directly update state. Ex: color = "red" is not allowed //

:Use a button to update the state //

```
;"import { useState } from "react" //
;"import ReactDOM from "react-dom/client" //

} ()function FavoriteColor //
;"const [color, setColor] = useState("red" //

) return //
<> //
<h1>My favorite color is {color}!</h1> //
button> //
"type="button //
{"onClick={() => setColor("blue" //
<Blue</button< //
</> //
( //
{ //

;"const root = ReactDOM.createRoot(document.getElementById('root' //
;"</ root.render(<FavoriteColor //
```

What Can State Hold //

The useState Hook can be used to keep track of strings, numbers, booleans, arrays, //
!objects, and any combination of these

.We could create multiple state Hooks to track individual values //

:Create multiple state Hooks //

```
;"import { useState } from "react" //
;"import ReactDOM from "react-dom/client" //

} ()function Car //
;"const [brand, setBrand] = useState("Ford" //
;"const [model, setModel] = useState("Mustang" //
;"const [year, setYear] = useState("1964" //
;"const [color, setColor] = useState("red" //

) return //
<> //
<h1>My {brand}</h1> //
<p> //
.{It is a {color} {model} from {year" //
```

```

<p/>    //
</>    //
(    //
{    //

```

```

;('const root = ReactDOM.createRoot(document.getElementById('root //
;</ root.render(<Car //

```

!Or, we can just use one state and include an object instead //

:Create a single Hook that holds an object //

```

;"import { useState } from "react //
;"import ReactDOM from "react-dom/client //

```

```

} ()function Car //
})const [car, setCar] = useState //
,"brand: "Ford    //
,"model: "Mustang  //
,"year: "1964     //
"color: "red      //
;{    //

) return //
<>    //
<h1>My {car.brand}</h1>    //
<p>    //
.{It is a {car.color} {car.model} from {car.year}    //
<p/>    //
</>    //
(    //
{    //

```

```

;('const root = ReactDOM.createRoot(document.getElementById('root //
;</ root.render(<Car //

```

Since we are now tracking a single object, we need to reference that object and then the //
(property of that object when rendering the component. (Ex: car.brand

Updating Objects and Arrays in State //

.When state is updated, the entire state gets overwritten //

?What if we only want to update the color of our car //

If we only called setCar({color: "blue"}), this would remove the brand, model, and year from //
.our state

We can use the JavaScript spread operator to help us //

:Use the JavaScript spread operator to update only the color of the car //

```
;"import { useState } from "react" //
;"import ReactDOM from "react-dom/client" //

} ()function Car //
})const [car, setCar] = useState //
,"brand: "Ford" //
,"model: "Mustang" //
,"year: "1964" //
"color: "red" //
;({ //

} <= () = const updateColor //
} <= setCar(previousState //
{"return { ...previousState, color: "blue" //
;({ //
{ //

) return //
<> //
<h1>My {car.brand}</h1> //
<p> //
.{It is a {car.color} {car.model} from {car.year} //
<p/> //
button> //
"type="button" //
{onClick={updateColor //
<Blue</button< //
</> //
( //
{ //

;((const root = ReactDOM.createRoot(document.getElementById('root' //
;(</ root.render(<Car //
```

Because we need the current value of state, we pass a function into our setCar function. //
.This function receives the previous value

.We then return an object, spreading the previousState and overwriting only the color //

React useEffect Hooks //

.The useEffect Hook allows you to perform side effects in your components //

.Some examples of side effects are: fetching data, directly updating the DOM, and timers //

.useEffect accepts two arguments. The second argument is optional //

(<useEffect(<function>, <dependency //

.Let's use a timer as an example //

:Use setTimeout() to count 1 second after initial render //

```
;"import { useState, useEffect } from "react //
```

```
;"import ReactDOM from "react-dom/client //
```

```
} ()function Timer //
```

```
;(const [count, setCount] = useState(0 //
```

```
} <= ())useEffect //
```

```
} <= ())setTimeout //
```

```
;(setCount((count) => count + 1 //
```

```
;(1000 ,{ //
```

```
;({// //
```

```
;<return <h1>I have rendered {count} times!</h1 //
```

```
{ //
```

```
;(('const root = ReactDOM.createRoot(document.getElementById('root //
```

```
;</ root.render(<Timer //
```

!But wait!! It keeps counting even though it should only count once //

useEffect runs on every render. That means that when the count changes, a render //
.happens, which then triggers another effect

.This is not what we want. There are several ways to control when side effects run //

We should always include the second parameter which accepts an array. We can //
.optionally pass dependencies to useEffect in this array

:No dependency passed .1 //

```
} <= ())useEffect //
```

```
Runs on every render// //
```

```
;({// //
```

:An empty array .2 //

```
} <= ())useEffect //
```

```
Runs only on the first render// //
```

```
;([] ,{ //
```


:Props or state values .3 //

```
} <= ())useEffect //
```

Runs on the first render// //

And any time any dependency value changes// //

```
;([prop, state] ,{ //
```

.So, to fix this issue, let's only run this effect on the initial render //

:Only run the effect on the initial render //

```
;"import { useState, useEffect } from "react //
```

```
;"import ReactDOM from "react-dom/client //
```

```
} ()function Timer //
```

```
;(const [count, setCount] = useState(0 //
```

```
} <= ())useEffect //
```

```
} <= ())setTimeout //
```

```
;(setCount((count) => count + 1 //
```

```
;(1000 ,{ //
```

add empty brackets here -> // ;([],{ //

```
;<return <h1>I've rendered {count} times!</h1 //
```

```
{ //
```

```
;(const root = ReactDOM.createRoot(document.getElementById('root //
```

```
;</ root.render(<Timer //
```

Here is an example of a useEffect Hook that is dependent on a variable. If the count //
:variable updates, the effect will run again

```
;"import { useState, useEffect } from "react //
```

```
;"import ReactDOM from "react-dom/client //
```

```
} ()function Counter //
```

```
;(const [count, setCount] = useState(0 //
```

```
;(const [calculation, setCalculation] = useState(0 //
```

```
} <= ())useEffect //
```

```
;(setCalculation(() => count * 2 //
```

```
count]); // <- add the count variable here] ,{ //
```

```
) return //
```

```
<> //
```

```
<p>Count: {count}</p> //
```

```
<button onClick={() => setCount((c) => c + 1)}>+</button> //
```

```
<p>Calculation: {calculation}</p> //
```

```
</> //  
;( //  
{ //
```

```
;(('const root = ReactDOM.createRoot(document.getElementById('root //  
;</ root.render(<Counter //
```

If there are multiple dependencies, they should be included in the useEffect dependency //
.array

Effect Cleanup //
.Some effects require cleanup to reduce memory leaks //

Timeouts, subscriptions, event listeners, and other effects that are no longer needed //
.should be disposed

.We do this by including a return function at the end of the useEffect Hook //

:Clean up the timer at the end of the useEffect Hook //

```
;"import { useState, useEffect } from "react //  
;"import ReactDOM from "react-dom/client //
```

```
} ()function Timer //  
;(const [count, setCount] = useState(0 //
```

```
} <= ())useEffect //  
} <= ())let timer = setTimeout //  
;(setCount((count) => count + 1 //  
;(1000 ,{ //
```

```
(return () => clearTimeout(timer //  
;([],{ //
```

```
;<return <h1>I've rendered {count} times!</h1 //  
{ //
```

```
;(('const root = ReactDOM.createRoot(document.getElementById('root //  
;</ root.render(<Timer //
```

```
*/  
.Note: To clear the timer, we had to name it  
/*
```

.Note: To clear the timer, we had to name it //

React useContext Hook //

React Context //

.React Context is a way to manage state globally //

It can be used together with the useState Hook to share state between deeply nested //

.components more easily than with useState alone

The Problem //

State should be held by the highest parent component in the stack that requires access to //

.the state

To illustrate, we have many nested components. The component at the top and bottom of //

.the stack need access to the state

To do this without Context, we will need to pass the state as "props" through each nested //

."component. This is called "prop drilling

:Passing "props" through nested components //

```
;"import { useState } from "react" //
```

```
;"import ReactDOM from "react-dom/client" //
```

```
} ()function Component1 //
```

```
;"const [user, setUser] = useState("Jesse Hall" //
```

```
) return //
```

```
<> //
```

```
<h1>{`Hello ${user}!`}</h1> //
```

```
</ {Component2 user={user} //
```

```
</> //
```

```
;( //
```

```
{ //
```

```
} ({ function Component2({ user //
```

```
) return //
```

```
<> //
```

```
<h1>Component 2</h1> //
```

```
</ {Component3 user={user} //
```

```
</> //
```

```
;( //
```

```
{ //
```

Even though components 2-4 did not need the state, they had to pass the state along so //

.that it could reach component 5

The Solution //

.The solution is to create context //

Create Context //

:To create context, you must Import createContext and initialize it //

```
;"import { useState, createContext } from "react" //
;"import ReactDOM from "react-dom/client" //
```

```
()const UserContext = createContext //
```

Next we'll use the Context Provider to wrap the tree of components that need the state //

.Context

Context Provider //

.Wrap child components in the Context Provider and supply the state value //

```
} ()function Component1 //
```

```
;"const [user, setUser] = useState("Jesse Hall" //
```

```
) return //
```

```
<{UserContext.Provider value={user}> //
```

```
<h1>{'Hello ${user}!'}</h1> //
```

```
</ {Component2 user={user}> //
```

```
<UserContext.Provider/> //
```

```
;( //
```

```
{ //
```

.Now, all components in this tree will have access to the user Context //

Use the useContext Hook //

In order to use the Context in a child component, we need to access it using the //

.useContext Hook

:First, include the useContext in the import statement //

```
;"import { useState, createContext, useContext } from "react" //
```

:Then you can access the user Context in all components //

```
} ()function Component5 //
```

```
;"const user = useContext(UserContext) //
```

```
) return //
```

```
<> //
```

```
<h1>Component 5</h1> //
```

```
<h2>{'Hello ${user} again!'}</h2> //
```

```
</> //
```

```
;( //
```

```
{ //
```

:Here is the full example using React Context //

```
;"import { useState, createContext, useContext } from "react" //
```

```
;"import ReactDOM from "react-dom/client" //
```

```

;()const UserContext = createContext //

} ()function Component1 //
;("const [user, setUser] = useState("Jesse Hall  //

) return //
<{UserContext.Provider value={user}> //
<h1>{'Hello ${user}!'}</h1> //
</ Component2> //
<UserContext.Provider/> //
;( //
{ //

} ()function Component2 //
) return //
<> //
<h1>Component 2</h1> //
</ Component3> //
</> //
;( //
{ //

} ()function Component3 //
) return //
<> //
<h1>Component 3</h1> //
</ Component4> //
</> //
;( //
{ //

} ()function Component4 //
) return //
<> //
<h1>Component 4</h1> //
</ Component5> //
</> //
;( //
{ //

} ()function Component5 //
;(const user = useContext(UserContext  //

) return //
<> //
<h1>Component 5</h1> //
<h2>{'Hello ${user} again!'}</h2> //

```

```

</> //
;( //
{ //

;(('const root = ReactDOM.createRoot(document.getElementById('root //
;</ root.render(<Component1 //

```

React useRef Hook //

.The useRef Hook allows you to persist values between renders //

.It can be used to store a mutable value that does not cause a re-render when updated //

.It can be used to access a DOM element directly //

Does Not Cause Re-renders //

If we tried to count how many times our application renders using the useState Hook, we //
 .would be caught in an infinite loop since this Hook itself causes a re-render

.To avoid this, we can use the useRef Hook //

.Use useRef to track application renders //

```

;"import { useState, useEffect, useRef } from "react //
;"import ReactDOM from "react-dom/client //

```

```

} ()function App //
;('')const [inputValue, setInputValue] = useState //
;(const count = useRef(0 //

} <= ())useEffect //
;count.current = count.current + 1 //
;{ //

) return //
<> //
input> //
"type="text //
{value={inputValue //
{(onChange={(e) => setInputValue(e.target.value //
</ //
<h1>Render Count: {count.current}</h1> //
</> //
;( //
{ //

```

```

;(('const root = ReactDOM.createRoot(document.getElementById('root //
;</ root.render(<App //

```

*/

Try typing in the input field, and you will
.see the application render count increase

/*

.useRef() only returns one item. It returns an Object called current //

.(When we initialize useRef we set the initial value: useRef(0 //

It's like doing this: const count = {current: 0}. We can access the count by using //
.count.current

Run this on your computer and try typing in the input to see the application render count //
.increase

Accessing DOM Elements //

.In general, we want to let React handle all DOM manipulation //

.But there are some instances where useRef can be used without causing issues //

.In React, we can add a ref attribute to an element to access it directly in the DOM //

:Use useRef to focus the input //

```
;"import { useRef } from "react" //  
;"import ReactDOM from "react-dom/client" //
```

```
} ()function App //  
;()const inputElement = useRef //
```

```
} <= () = const focusInput //  
;()inputElement.current.focus //  
;{ //
```

```
) return //  
<> //  
</ {input type="text" ref={inputElement} //  
<button onClick={focusInput}>Focus Input</button> //  
</> //  
;{ //  
{ //
```

```
;('const root = ReactDOM.createRoot(document.getElementById('root' //  
;</ root.render(<App //
```

*/

,By clicking the button

.the input field will get focus
/*

Tracking State Changes //

.The useRef Hook can also be used to keep track of previous state values //

.This is because we are able to persist useRef values between renders //

:Use useRef to keep track of previous state values //

```
;"import { useState, useEffect, useRef } from "react" //  
;"import ReactDOM from "react-dom/client" //
```

```
} ()function App //  
;"(")const [inputValue, setInputValue] = useState //  
;"(")const previousInputValue = useRef //
```

```
} <= ()useEffect //  
;previousInputValue.current = inputValue //  
;([inputValue] ,{ //
```

```
) return //  
<> //  
input> //  
"type="text //  
{value={inputValue //  
{(onChange={e => setInputValue(e.target.value //  
</ //  
<h2>Current Value: {inputValue}</h2> //  
<h2>Previous Value: {previousInputValue.current}</h2> //  
</> //  
;( //  
{ //
```

```
;'('const root = ReactDOM.createRoot(document.getElementById('root' //  
;(</ root.render(<App //
```

*/

Start typing in the input field and you will

.see the current- and previous value

/*

This time we use a combination of useState, useEffect, and useRef to keep track of the //
.previous state

In the useEffect, we are updating the useRef current value each time the inputValue is //
.updated by entering text into the input field

React useReducer Hook //

.The useReducer Hook is similar to the useState Hook //

.It allows for custom state logic //

If you find yourself keeping track of multiple pieces of state that rely on complex logic, //
.useReducer may be useful

Syntax //

.The useReducer Hook accepts two arguments //

(<useReducer(<reducer>, <initialState //

The reducer function contains your custom state logic and the initialState can be a simple //
.value but generally will contain an object

.The useReducer Hook returns the current state and a dispatch method //

:Here is an example of useReducer in a counter app //

```
;"import { useReducer } from "react" //  
;"import ReactDOM from "react-dom/client" //
```

```
] = const initialTodos //  
} //  
,id: 1 //  
,title: "Todo 1 //  
,complete: false //  
,{ //  
} //  
,id: 2 //  
,title: "Todo 2 //  
,complete: false //  
,{ //  
:[ //
```

```
} <= (const reducer = (state, action //  
} (switch (action.type //  
:"case "COMPLETE //  
} <= (return state.map((todo //  
} (if (todo.id === action.id //  
;{ return { ...todo, complete: !todo.complete //  
} else { //  
;return todo //  
{ //  
;{ //  
:default //
```

```

;return state    //
{ //
;{ //

} ()function Todos //
;(const [todos, dispatch] = useReducer(reducer, initialTodos //

} <= (const handleComplete = (todo //
;{ dispatch({ type: "COMPLETE", id: todo.id //
;{ //

) return //
<> //
) <= (todos.map((todo) //
<{div key={todo.id} //
<label> //
input> //
"type="checkbox //
{checked={todo.complete //
{(onChange={() => handleComplete(todo //
</ //
{todo.title} //
<label/> //
<div/> //
{(( //
</> //
;( //
{ //

;('const root = ReactDOM.createRoot(document.getElementById('root //
;(</ root.render(<Todos //

```

.This is just the logic to keep track of the todo complete status //

All of the logic to add, delete, and complete a todo could be contained within a single //
.useReducer Hook by adding more actions

React useCallback Hook //

.The React useCallback Hook returns a memoized callback function //

.Think of memoization as caching a value so that it does not need to be recalculated //

This allows us to isolate resource intensive functions so that they will not automatically run //
.on every render

.The useCallback Hook only runs when one of its dependencies update //

.This can improve performance //

The useCallback and useMemo Hooks are similar. The main difference is that useMemo // returns a memoized value and useCallback returns a memoized function. You can learn .more about useMemo in the useMemo chapter

Problem //

One reason to use useCallback is to prevent a component from re-rendering unless its // .props have changed

In this example, you might think that the Todos component will not re-render unless the // :todos change

index.js //

index.js //

```
;"import { useState } from "react" //
;"import ReactDOM from "react-dom/client" //
;"import Todos from "./Todos" //
```

```
} <= () = const App //
;(const [count, setCount] = useState(0 //
;([[]])const [todos, setTodos] = useState //
```

```
} <= () = const increment //
;(setCount((c) => c + 1 //
;{ //
} <= () = const addTodo //
;(["setTodos((t) => [...t, "New Todo //
;{ //
```

```
) return //
<> //
</ {Todos todos={todos} addTodo={addTodo}> //
</ hr> //
<div> //
{Count: {count //
<button onClick={increment}>+</button> //
<div/> //
</> //
;{ //
;{ //
```

```
;(('const root = ReactDOM.createRoot(document.getElementById('root' //
;(</ root.render(<App //
```

Todos.js //

```

;import { useState } from "react" //
;import ReactDOM from "react-dom/client" //
;import Todos from "../Todos" //

} <= () = const App //
;(const [count, setCount] = useState(0 //
;([])const [todos, setTodos] = useState //

} <= () = const increment //
;(setCount((c) => c + 1 //
;{ //
} <= () = const addTodo //
;(["setTodos((t) => [...t, "New Todo //
;{ //

) return //
<> //
</ {Todos todos={todos} addTodo={addTodo} //
</ hr> //
<div> //
{Count: {count //
<button onClick={increment}>+</button> //
<div/> //
</> //
;{ //
;{ //

;((const root = ReactDOM.createRoot(document.getElementById('root //
;(</ root.render(<App //

```

.Try running this and click the count increment button //

.You will notice that the Todos component re-renders even when the todos do not change //

Why does this not work? We are using memo, so the Todos component should not //
re-render since neither the todos state nor the addTodo function are changing when the
.count is incremented

."This is because of something called "referential equality" //

Every time a component re-renders, its functions get recreated. Because of this, the //
.addTodo function has actually changed

Solution //

To fix this, we can use the useCallback hook to prevent the function from being recreated //
.unless necessary

:Use the useCallback Hook to prevent the Todos component from re-rendering needlessly //

index.js //

```
;"import { useState, useCallback } from "react" //
;"import ReactDOM from "react-dom/client" //
;"import Todos from "../Todos" //

} <= () = const App //
;(const [count, setCount] = useState(0 //
;([])const [todos, setTodos] = useState //

} <= () = const increment //
;(setCount((c) => c + 1 //
;{ //
} <= ())const addTodo = useCallback //
;(["setTodos((t) => [...t, "New Todo //
;([todos] ,{ //

) return //
<> //
</ {Todos todos={todos} addTodo={addTodo}> //
</ hr> //
<div> //
{Count: {count //
<button onClick={increment}>+</button> //
<div/> //
</> //
;{ //
;{ //

;((const root = ReactDOM.createRoot(document.getElementById('root //
;(</ root.render(<App //
```

Todos.js //

```
;"import { useState, useCallback } from "react" //
;"import ReactDOM from "react-dom/client" //
;"import Todos from "../Todos" //

} <= () = const App //
;(const [count, setCount] = useState(0 //
;([])const [todos, setTodos] = useState //

} <= () = const increment //
;(setCount((c) => c + 1 //
;{ //
} <= ())const addTodo = useCallback //
;(["setTodos((t) => [...t, "New Todo //
```

```

;([todos] ,{ //

) return //
<> //
</ {Todos todos={todos} addTodo={addTodo} //
</ hr> //
<div> //
{Count: {count //
<button onClick={increment}>+</button> //
<div/> //
</> //
;( //
;{ //

;('const root = ReactDOM.createRoot(document.getElementById('root //
;(</ root.render(<App //

```

React useMemo Hook //

.The React useMemo Hook returns a memoized value //

.Think of memoization as caching a value so that it does not need to be recalculated //

.The useMemo Hook only runs when one of its dependencies update //

.This can improve performance //

The useMemo and useCallback Hooks are similar. The main difference is that useMemo //
returns a memoized value and useCallback returns a memoized function. You can learn
.more about useCallback in the useCallback chapter

Performance //

The useMemo Hook can be used to keep expensive, resource intensive functions from //
.needlessly running

.In this example, we have an expensive function that runs on every render //

.When changing the count or adding a todo, you will notice a delay in execution //

:A poor performing function. The expensiveCalculation function runs on every render //

```

;"import { useState } from "react //
;"import ReactDOM from "react-dom/client //

```

```

} <= () = const App //
;(const [count, setCount] = useState(0 //
;([])const [todos, setTodos] = useState //
;(const calculation = expensiveCalculation(count //

```

```

} <= () = const increment //
;(setCount((c) => c + 1 //
;{ //
} <= () = const addTodo //
;(["setTodos((t) => [...t, "New Todo //
;{ //

) return //
<div> //
<div> //
<h2>My Todos</h2> //
} <= (todos.map((todo, index) //
;<return <p key={index}>{todo}</p //
;{{ //
<button onClick={addTodo}>Add Todo</button> //
<div/> //
</ hr> //
<div> //
{Count: {count //
<button onClick={increment}>+</button> //
<h2>Expensive Calculation</h2> //
{calculation} //
<div/> //
<div/> //
;( //
;{ //

} <= (const expensiveCalculation = (num //
;("...console.log("Calculating //
} (++for (let i = 0; i < 1000000000; i //
;num += 1 //
{ //
;return num //
;{ //

;(const root = ReactDOM.createRoot(document.getElementById('root //
;</ root.render(<App //

```

Use useMemo //

To fix this performance issue, we can use the useMemo Hook to memoize the //
.expensiveCalculation function. This will cause the function to only run when needed

.We can wrap the expensive function call with useMemo //

The useMemoHook accepts a second parameter to declare dependencies. The expensive //
.function will only run when its dependencies have changed

In the following example, the expensive function will only run when count is changed and // not when todo's are added

:Performance example using the useMemo Hook //

```
;"import { useState, useMemo } from "react" //
;"import ReactDOM from "react-dom/client" //

} <= () = const App //
;(const [count, setCount] = useState(0 //
;([])const [todos, setTodos] = useState //
;([const calculation = useMemo(() => expensiveCalculation(count), [count //

} <= () = const increment //
;(setCount((c) => c + 1 //
;{ //
} <= () = const addTodo //
;(["setTodos((t) => [...t, "New Todo //
;{ //

) return //
<div> //
<div> //
<h2>My Todos</h2> //
} <= (todos.map((todo, index) //
;<return <p key={index}>{todo}</p //
{{ //
<button onClick={addTodo}>Add Todo</button> //
<div/> //
</ hr> //
<div> //
{Count: {count //
<button onClick={increment}>+</button> //
<h2>Expensive Calculation</h2> //
{calculation} //
<div/> //
<div/> //
;( //
;{ //

} <= (const expensiveCalculation = (num //
;"...console.log("Calculating //
} (++for (let i = 0; i < 1000000000; i //
;num += 1 //
{ //
;return num //
;{ //
```



```
;('const root = ReactDOM.createRoot(document.getElementById('root //
;</ root.render(<App //
```

React Custom Hooks //

.Hooks are reusable functions //

When you have component logic that needs to be used by multiple components, we can //
.extract that logic to a custom Hook

.Custom Hooks start with "use". Example: useFetch //

Build a Hook //

.In the following code, we are fetching data in our Home component and displaying it //

We will use the JSONPlaceholder service to fetch fake data. This service is great for //
.testing applications when there is no existing data

.To learn more, check out the JavaScript Fetch API section //

Use the JSONPlaceholder service to fetch fake "todo" items and display the titles on the //
:page

:index.js //

```
;"import { useState, useEffect } from "react //
;"import ReactDOM from "react-dom/client //
```

```
} <= () = const Home //
;(const [data, setData] = useState(null //
```

```
} <= ())useEffect //
("fetch("https://jsonplaceholder.typicode.com/todos //
(()then((res) => res.json. //
;((then((data) => setData(data. //
;([],{ //
```

```
) return //
<> //
&& data} //
} <= (data.map((item //
;<return <p key={item.id}>{item.title}</p //
{({ //
</> //
;( //
;{ //
```

```
;('const root = ReactDOM.createRoot(document.getElementById('root //
```

```
;</ root.render(<Home //
```

The fetch logic may be needed in other components as well, so we will extract that into a //
.custom Hook

:Move the fetch logic to a new file to be used as a custom Hook //

:useFetch.js //

```
;"import { useState, useEffect } from "react //
```

```
} <= (const useFetch = (url //  
;(const [data, setData] = useState(null //
```

```
} <= ())(useEffect //  
(fetch(url //  
(()then((res) => res.json. //  
;((then((data) => setData(data. //  
;([url] ,{ //
```

```
:[return [data //  
:{ //
```

```
;export default useFetch //
```

:index.js //

```
;"import ReactDOM from "react-dom/client //  
;"import useFetch from "../useFetch //
```

```
} <= () = const Home //  
;"const [data] = useFetch("https://jsonplaceholder.typicode.com/todos //
```

```
) return //  
<> //  
&& data} //  
} <= (data.map((item //  
;<return <p key={item.id}>{item.title}</p //  
{{ //  
</> //  
;( //  
:{ //
```

```
;(const root = ReactDOM.createRoot(document.getElementById('root //  
;</ root.render(<Home //
```

Example Explained //

We have created a new file called useFetch.js containing a function called useFetch which //
.contains all of the logic needed to fetch our data

We removed the hard-coded URL and replaced it with a url variable that can be passed to //
.the custom Hook

.Lastly, we are returning our data from our Hook //

In index.js, we are importing our useFetch Hook and utilizing it like any other Hook. This is //
.where we pass in the URL to fetch data from

.Now we can reuse this custom Hook in any component to fetch data from any URL //

//*****END***** //