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.

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](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!

### **Example**

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>

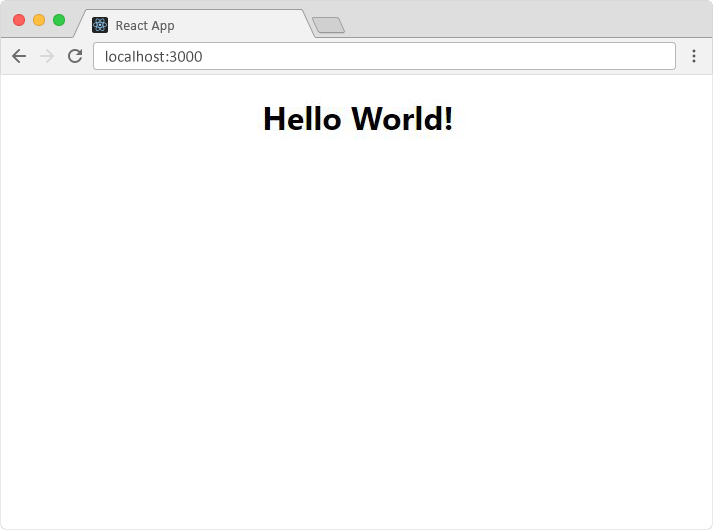
);

}

export default App;

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

The result:



## 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:

### **Example**

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);

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](https://www.w3schools.com/react/react_es6_classes.asp)
* [Arrow Functions](https://www.w3schools.com/react/react_es6_arrow.asp)
* [Variables](https://www.w3schools.com/react/react_es6_variables.asp) (let, const, var)
* [Array Methods](https://www.w3schools.com/react/react_es6_array_methods.asp) like .map()
* [Destructuring](https://www.w3schools.com/react/react_es6_destructuring.asp)
* [Modules](https://www.w3schools.com/react/react_es6_modules.asp)
* [Ternary Operator](https://www.w3schools.com/react/react_es6_ternary.asp)
* [Spread Operator](https://www.w3schools.com/react/react_es6_spread.asp)

## 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.

### **Example**[**Get your own React.js Server**](https://www.w3schools.com/spaces/)

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:



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

## Method in Classes

You can add your own methods in a class:

### **Example**

Create a method named "present":

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

## Class Inheritance

To create a class inheritance, use the extends keyword.

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

### **Example**

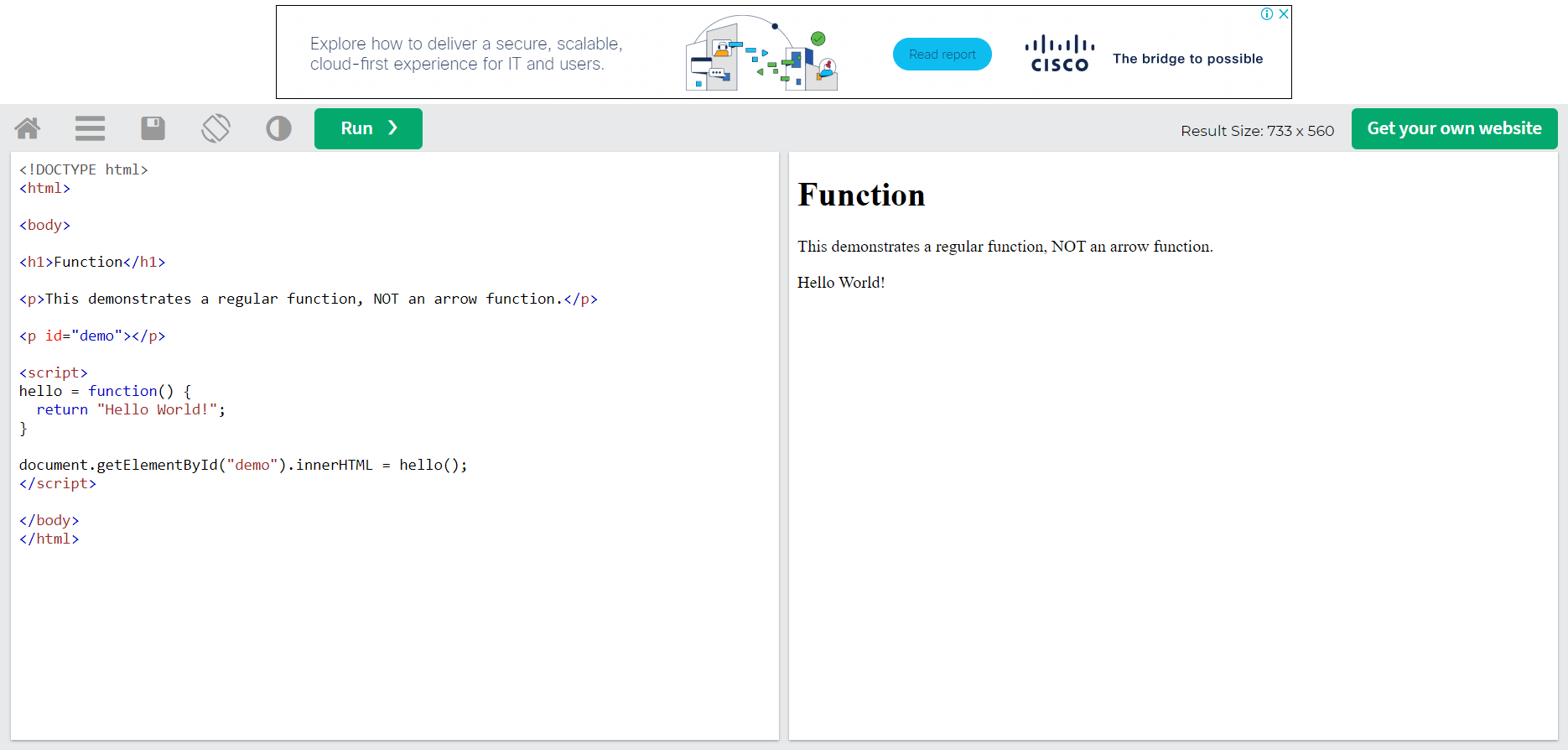
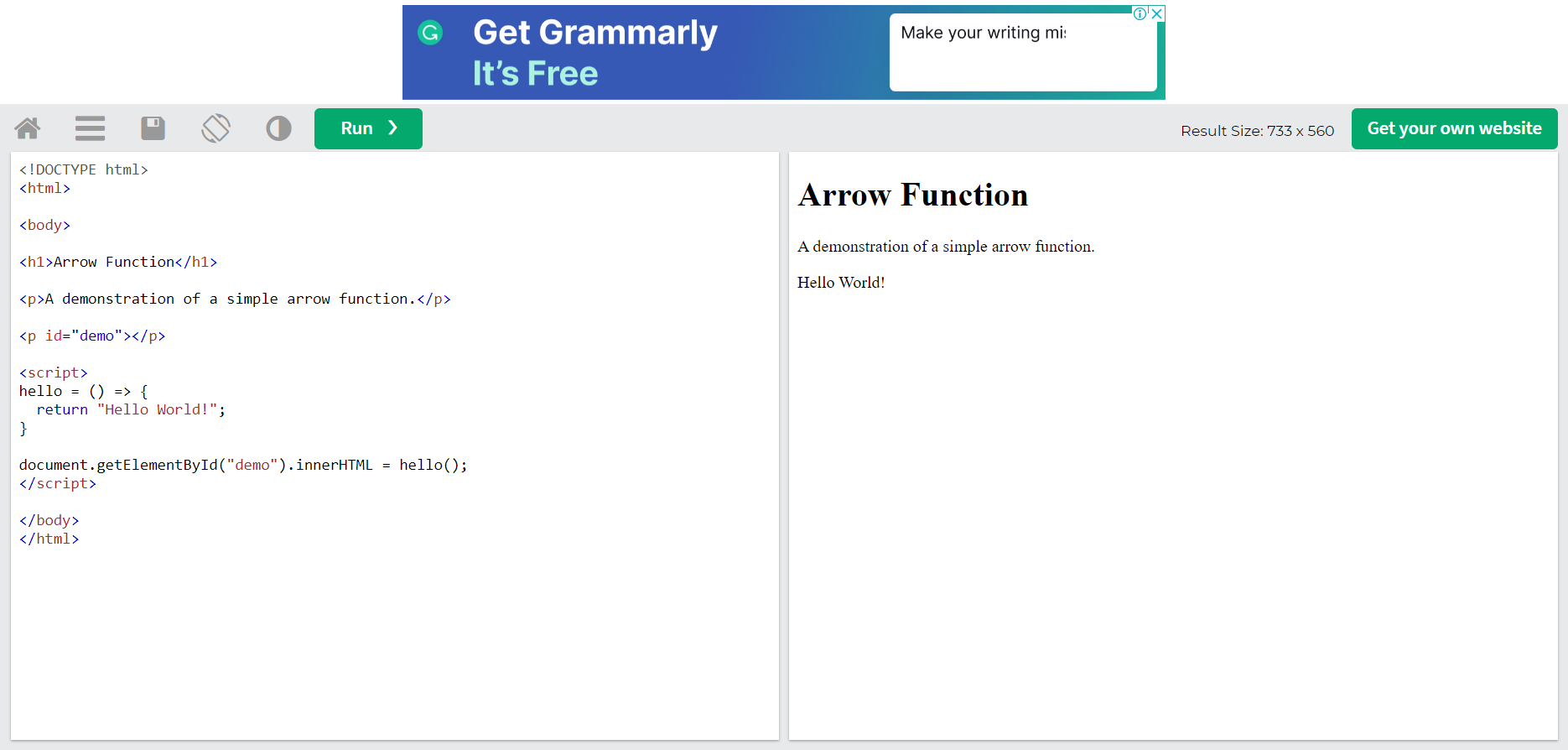
Create a class named "Model" which will inherit the methods from the "Car" class:

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.

## Arrow Functions

Arrow functions allow us to write shorter function syntax:



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:

### **Example**

Arrow Functions Return Value by Default:



. 

## 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.





## 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.

### **Example**[**Get your own React.js Server**](https://www.w3schools.com/spaces/)

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.

### **Example**

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.

### **Example**

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

## 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.

### **Example**[**Get your own React.js Server**](https://www.w3schools.com/spaces/)

Generate a list of items from an array:

const myArray = ['apple', 'banana', 'orange'];

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

## 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:

### **Example**[**Get your own React.js Server**](https://www.w3schools.com/spaces/)

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:

### **Example**

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:

### **Example**

## Destructuring Objects

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

### **Example**

Before:

const vehicleOne = {

brand: 'Ford',

model: 'Mustang',

type: 'car',

year: 2021,

color: 'red'

}

myVehicle(vehicleOne);

// old way

function myVehicle(vehicle) {

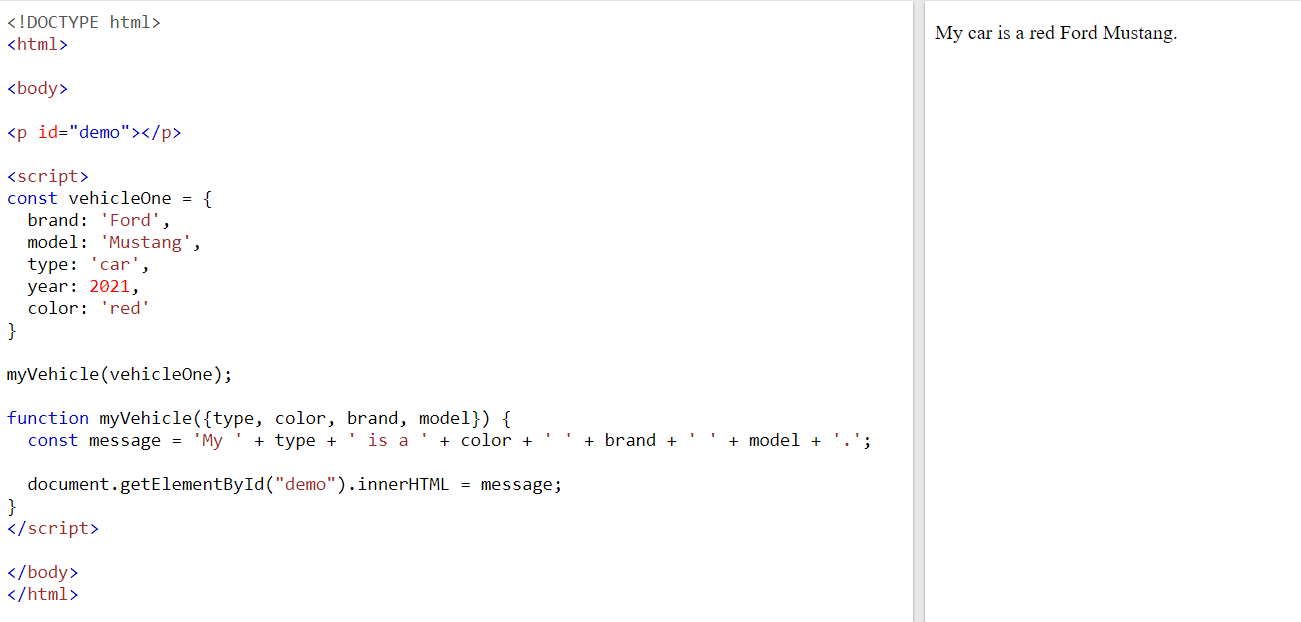
const message = 'My ' + vehicle.type + ' is a ' + vehicle.color + ' ' + vehicle.brand + ' ' + vehicle.model + '.';

}

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

### **Example**

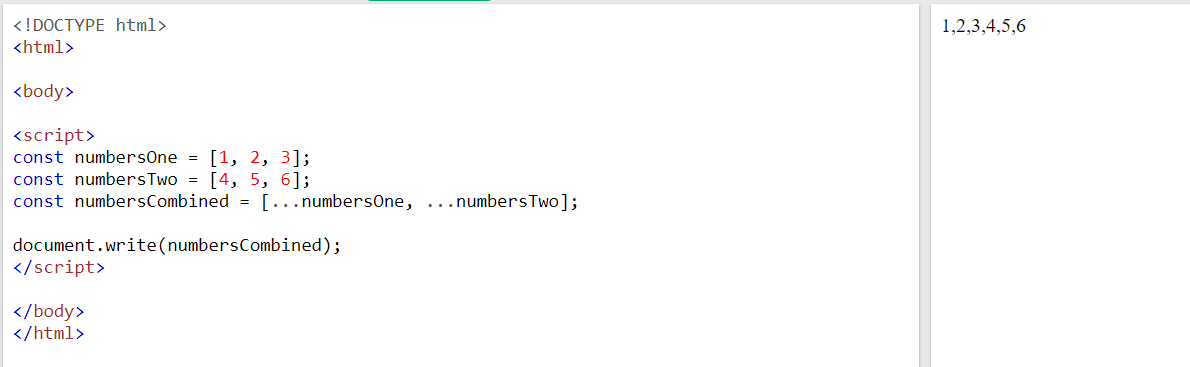
With destructuring:

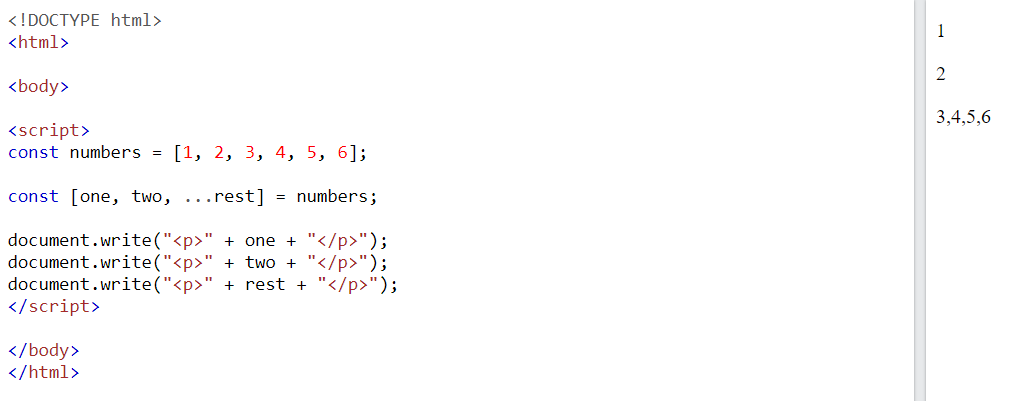


## Spread Operator

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

### **Example**





## 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.

### **Example**[**Get your own React.js Server**](https://www.w3schools.com/spaces/)

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.

### **Example**

message.js

const message = () => {

const name = "Jesse";

const age = 40;

return name + ' is ' + age + 'years old.';

};

export default message;

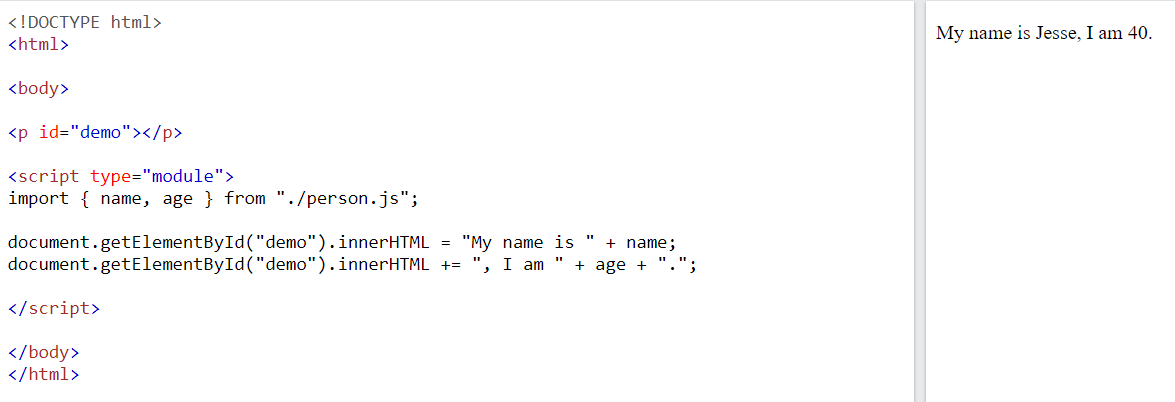
## Import

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.

### **Example**

Import named exports from the file person.js:



## Ternary Operator

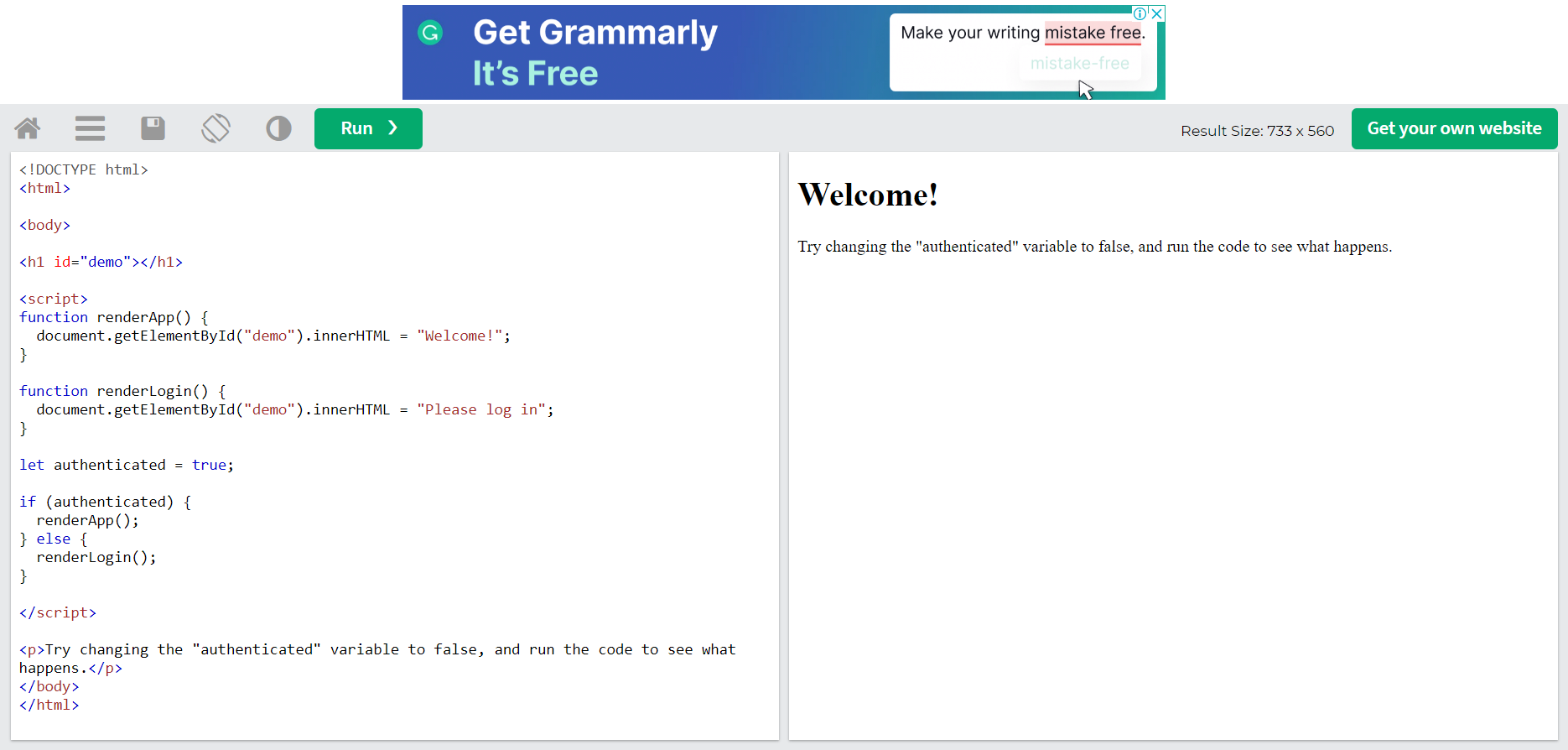
The ternary operator is a simplified conditional operator like if / else.

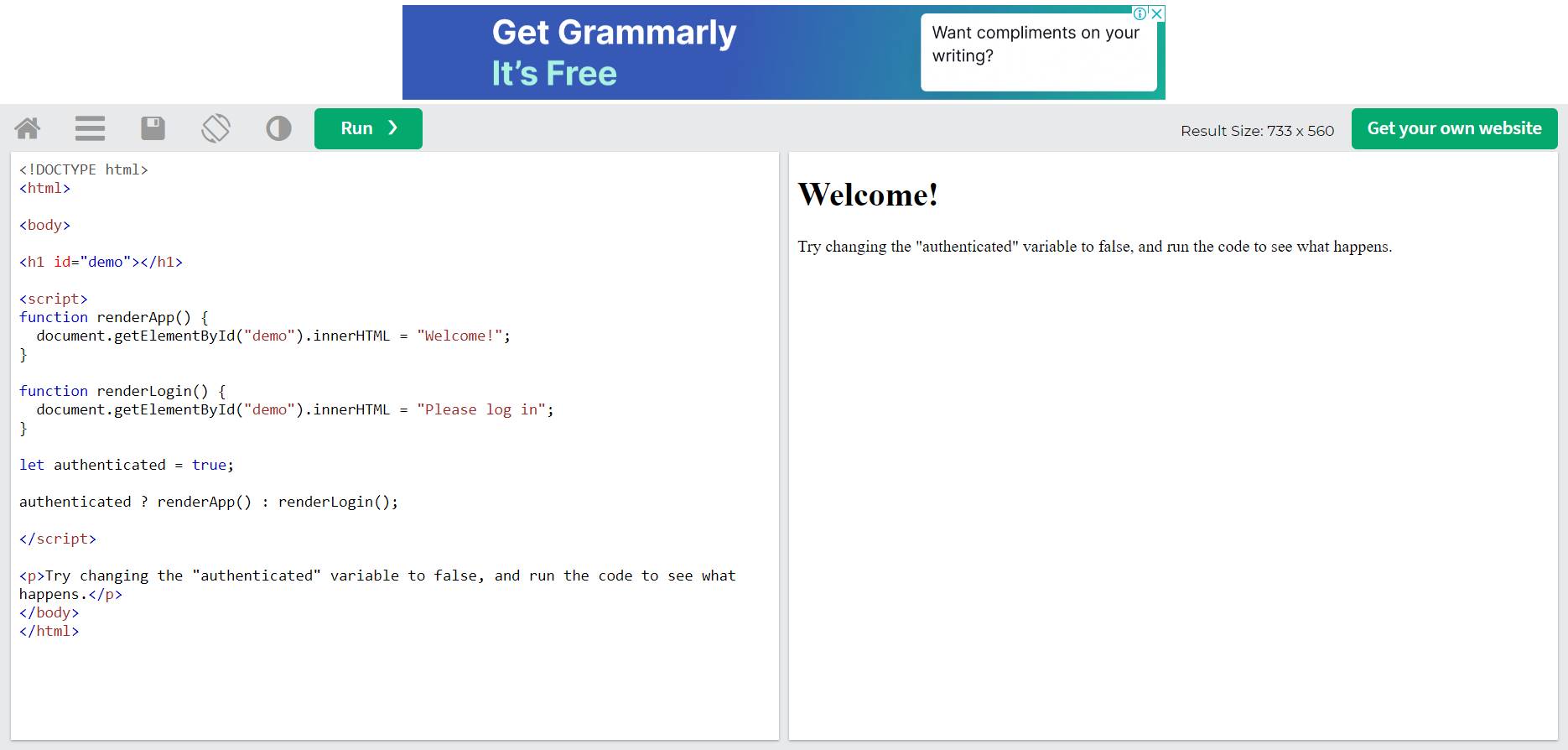
Syntax: condition ? <expression if true> : <expression if false>

Here is an example using if / else:

### **Example**[**Get your own React.js Server**](https://www.w3schools.com/spaces/)

Before:



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.

### **Example**[**Get your own React.js Server**](https://www.w3schools.com/spaces/)

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:

<body>

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

</body>

## 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.

### **Example**

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

## 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':

### **Example**

The root node can be called whatever you like:

<body>

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

</body>

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

## 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:

### **Example 1**[**Get your own React.js Server**](https://www.w3schools.com/spaces/)

JSX:

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

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

## Expressions in JSX

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

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

### **Example**

Execute the expression 5 + 5:

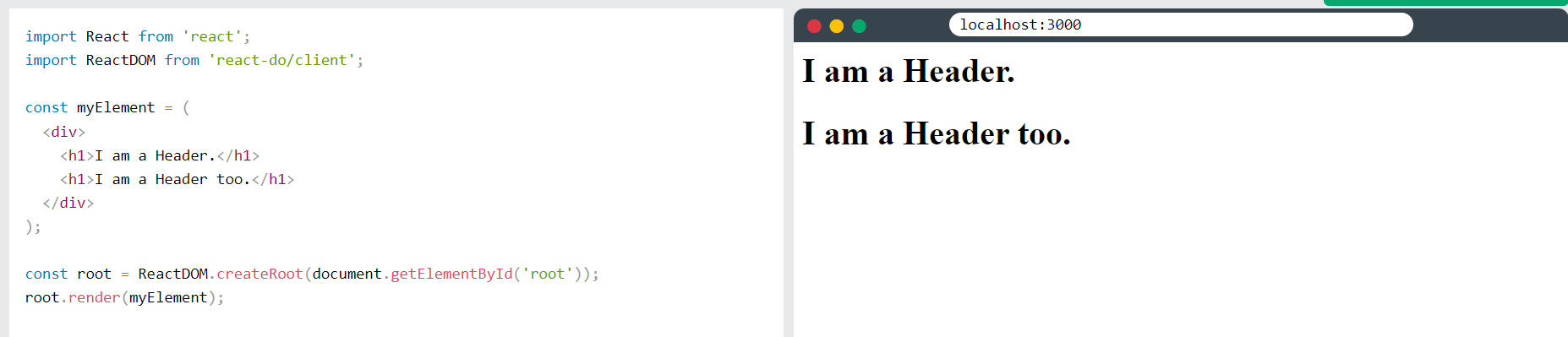
## Inserting a Large Block of HTML

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

## 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.

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.

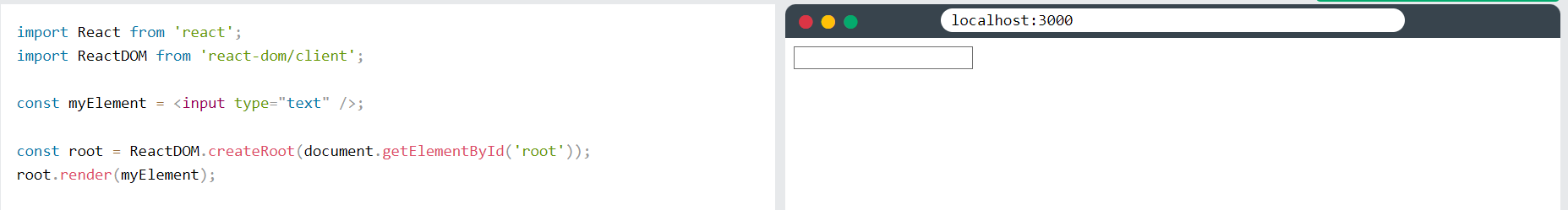
A fragment looks like an empty HTML tag: <></>.

## Elements Must be Closed

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

### **Example**

Close empty elements with />

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.

### **Example**

Use attribute className instead of class in JSX:

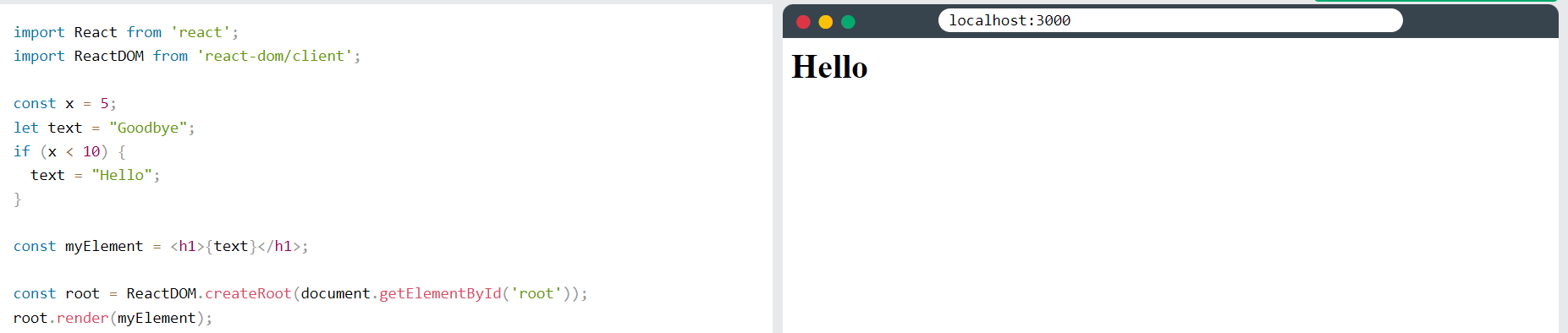
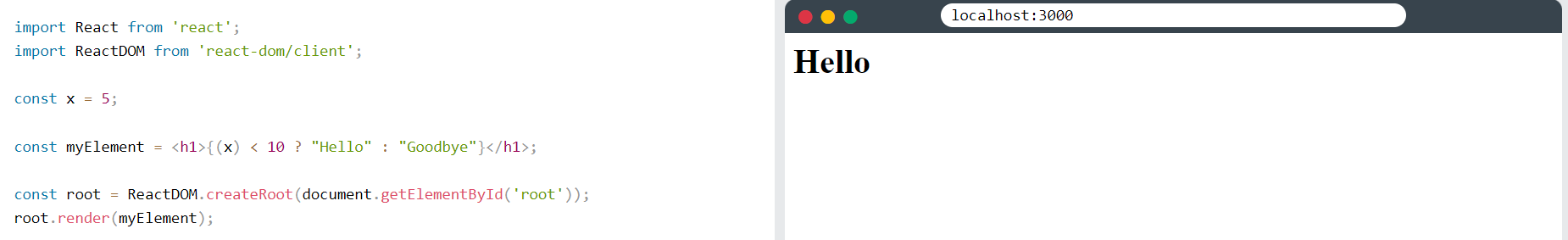
## Conditions - if statements

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:

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

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.

### **Example**[**Get your own React.js Server**](https://www.w3schools.com/spaces/)

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.

### **Example**

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 />

### **Example**

Display the Car component in the "root" element:

## 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.

### **Example**

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

## Components in Components

We can refer to components inside other components:

## 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.

### **Example**

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

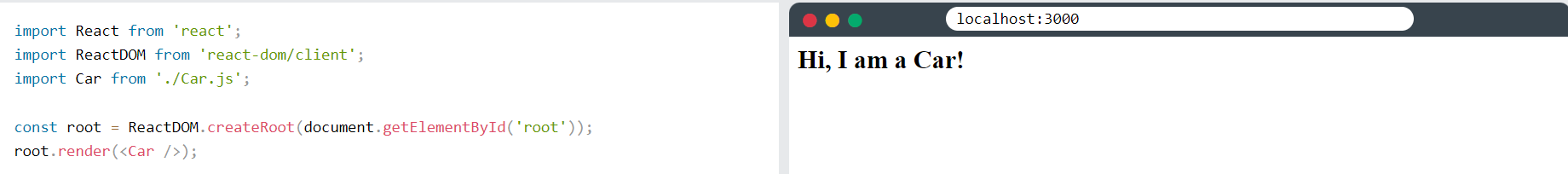
function Car() {

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.

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.

### **xample**[**Get your own React.js Server**](https://www.w3schools.com/spaces/)

Create a Class component called Car

class Car extends React.Component {

render() {

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

}

}

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

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

### **Example**

Display the Car component in the "root" element:

## Component Constructor

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

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

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

You will learn more about state later in this tutorial.

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

### **Example**

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

class Car extends React.Component {

constructor() {

super();

this.state = {color: "red"};

}

render() {

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

}

}

Use the color property in the render() function:

### **Example**

## Props

Another way of handling component properties is by using props.

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

You will learn more about props in the next chapter.

### **Example**

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

## Props in the Constructor

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

### **Example**

## Components in Components

We can refer to components inside other components:

## Components in Files

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

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

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

### **Example**

This is the new file, we named it 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.

### **Example**

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

## 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:

### **Example**

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:

### **Example**

Specify all the properties your component need:

class Car extends React.Component {

constructor(props) {

super(props);

this.state = {

brand: "Ford",

model: "Mustang",

color: "red",

year: 1964

};

}

render() {

return (

<div>

<h1>My Car</h1>

</div>

);

}

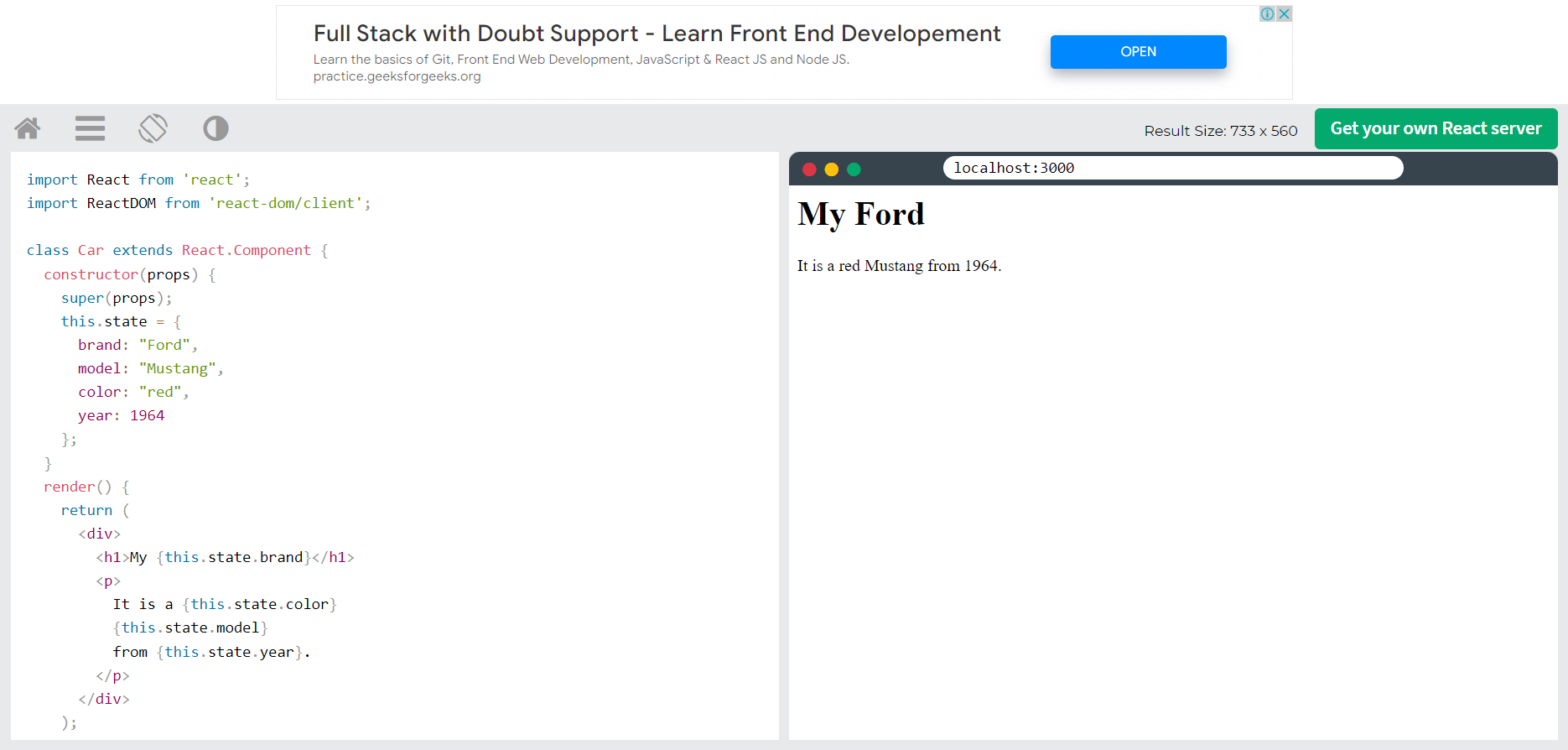
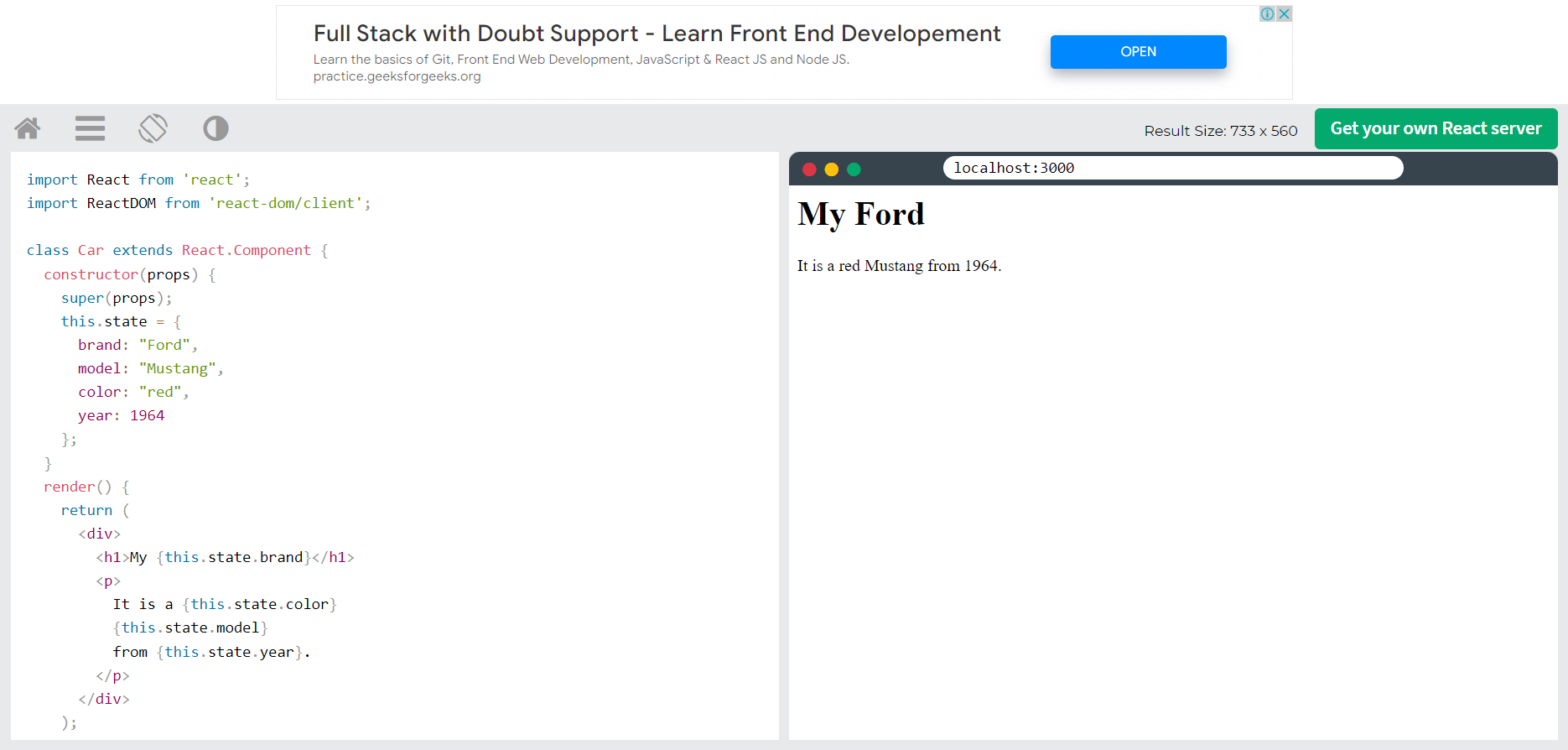
}

## Using the state Object

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

### **Example:**

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

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:

1. constructor()
2. getDerivedStateFromProps()
3. render()
4. 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).

### **Example:**

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

### **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:

### **Example:**

The getDerivedStateFromProps method is called right before the render method:

### **render**

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

### **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.

### **Example:**

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

## 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:

1. getDerivedStateFromProps()
2. shouldComponentUpdate()
3. render()
4. getSnapshotBeforeUpdate()
5. 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:

### **Example:**

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

### **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:

### **Example:**

Stop the component from rendering at any update:

### **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:

### **Example:**

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

### **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:

### **Example:**

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

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 />);

### **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:

### **Example:**

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

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:

### **Example:**

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

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 />);

[Run Example »](https://www.w3schools.com/react/showreact.asp?filename=demo2_react_lifecycle_componentdidupdate)

## 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.

### **Example:**

Click the button to delete the header:

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 />);

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:

### **Example**[**Get your own React.js Server**](https://www.w3schools.com/spaces/)

Add a "brand" attribute to the Car element:

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

The component receives the argument as a props object:

### **Example**

Use the brand attribute in the component:

## Pass Data

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

### **Example**

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

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 />);

[Run Example »](https://www.w3schools.com/react/showreact.asp?filename=demo2_react_props_pass)

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

### **Example**

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

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 />);

[Run Example »](https://www.w3schools.com/react/showreact.asp?filename=demo2_react_props_pass2)

Or if it was an object:

### **Example**

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

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 />);

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:**[**Get your own React.js Server**](https://www.w3schools.com/spaces/)

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

### **HTML:**

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

### **Example:**

Put the shoot function inside the Football component:

## Passing Arguments

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

### **Example:**

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

## React Event Object

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

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

### **Example:**

Arrow Function: Sending the event object manually:

