React Introduction

---

https://www.w3schools.com/react/react_intro.asp

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 V17.0.2 (August 2021).

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 v4.0.3 (August 2021).

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

React Getting Started

To use React in production, you need npm which is included with Node.js.

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.

Example

Include three CDN's in your HTML file:

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://unpkg.com/react@17/umd/react.development.js"
crossorigin></script>
    <script src="https://unpkg.com/react-dom@17/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**.

Get Certified!

Complete the React modules, do the exercises, take the exam and become w3schools certified!!

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:

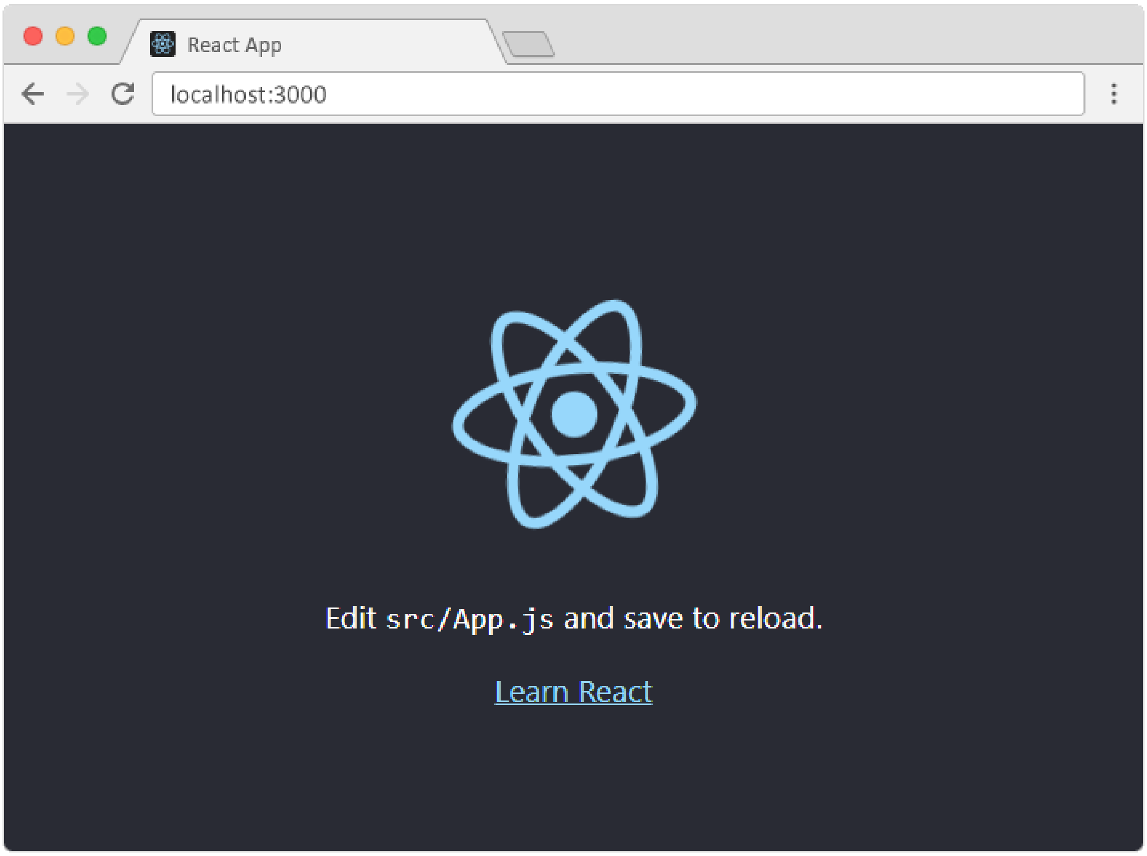
```
cd my-react-app
```

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.

The result:



Edit `src/App.js` and save to reload.

Learn React

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!

Example

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

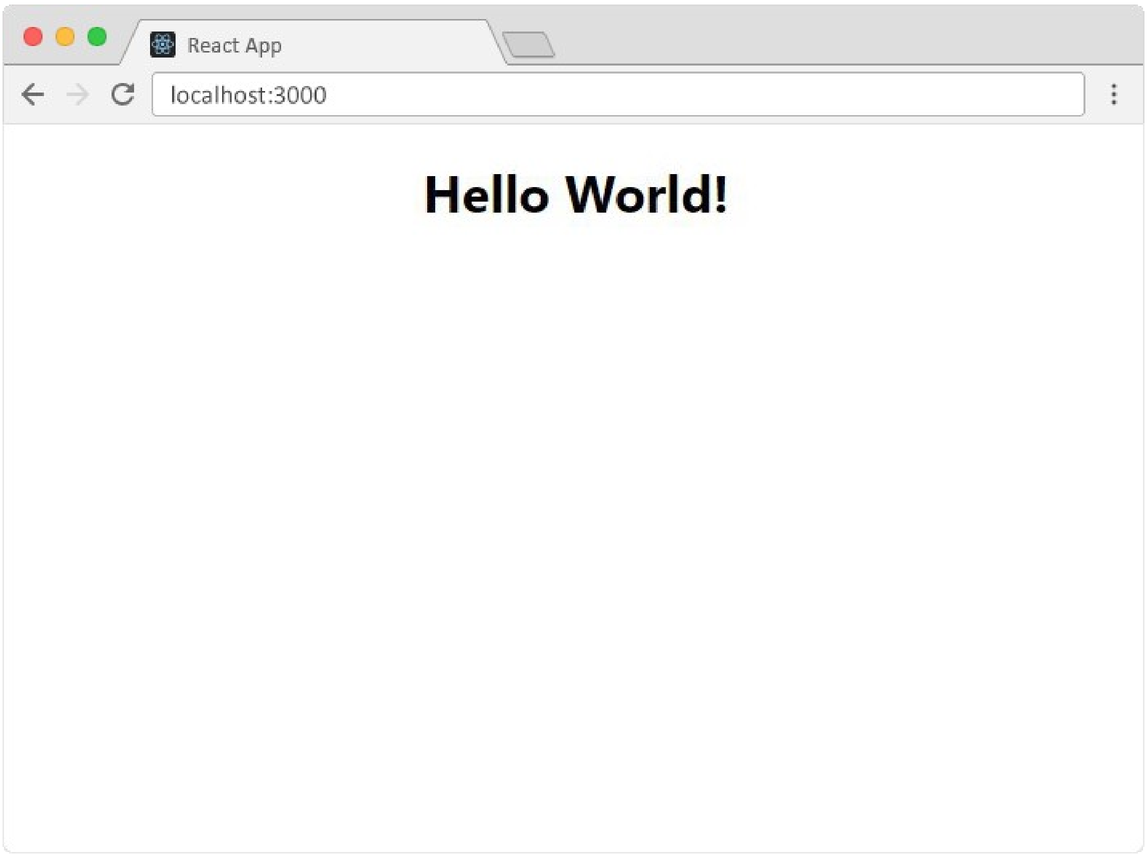
See the changes in the browser when you click Save.

```
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:



Hello World!

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';

const myfirstelement = <h1>Hello React!</h1>

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

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.

Example

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:

Example

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

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

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

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

Get Certified!

Complete the React modules, do the exercises, take the exam and become w3schools certified!!

Method in Classes

You can add your own methods in a class:

Example

Create a method named "present":

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

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

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

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

Class Inheritance

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

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

Example

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

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

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

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

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

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

To learn more about classes, check out our JavaScript Classes section.

React ES6 Arrow Functions

Arrow Functions

Arrow functions allow us to write shorter function syntax:

Before:

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

With Arrow Function:

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

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

Arrow Functions Return Value by Default:

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

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

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

Arrow Function With Parameters:

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

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

Arrow Function Without Parentheses:

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

What About `this`?

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

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

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

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

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

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

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

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

Example

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

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

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

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);
```

Example

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

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

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

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);
```

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

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

var

```
var x = 5.6;
```

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

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

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

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

let

```
let x = 5.6;
```

`let` 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.

Get Certified!

Complete the React modules, do the exercises, take the exam and become w3schools certified!!

---

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.

Example

Generate a list of items from an array:

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

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

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:

Example

```
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);
```

Get Certified!

Complete the React modules, do the exercises, take the exam and become w3schools certified!!

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:

```
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 +
'.';
}
```

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:

Example

```
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 + '.';
}
```

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.

Example

```
const numbersOne = [1, 2, 3];
const numbersTwo = [4, 5, 6];
const numbersCombined = [...numbersOne, ...numbersTwo];
```

The spread operator is often used in combination with destructuring.

Example

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

```
const numbers = [1, 2, 3, 4, 5, 6];

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

We can use the spread operator with objects too:

Example

Combine these two objects:

```
const myVehicle = {
  brand: 'Ford',
  model: 'Mustang',
  color: 'red'
}
```

```
const updateMyVehicle = {
  type: 'car',
  year: 2021,
  color: 'yellow'
}

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

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 }
```

Get Certified!

Complete the React modules, do the exercises, take the exam and become w3schools certified!!

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.

Import from named exports

Import named exports from the file person.js:

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

Import from default exports

Import a default export from the file message.js:

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

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:

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

Here is the same example using a ternary operator:

With Ternary

```
authenticated ? renderApp() : renderLogin();
```

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.

Example

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>
```

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

Get Certified!

Complete the React modules, do the exercises, take the exam and become w3schools certified!!

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:

```
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'`:

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:

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

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:

Example 1

JSX:

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

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

Example 2

Without JSX:

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

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

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

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

Get Certified!

Complete the React modules, do the exercises, take the exam and become w3schools certified!!

Expressions in JSX

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

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

Example

Execute the expression `5 + 5`:

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

Inserting a Large Block of HTML

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

Example

Create a list with three list items:

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

One Top Level Element

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

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

Example

Wrap two paragraphs inside one DIV element:

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

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: `<></>`.

Example

Wrap two paragraphs inside a fragment:

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

Elements Must be Closed

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

Example

Close empty elements with `/>`

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

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

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:

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

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:

Example

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

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

const myelement = <h1>{text}</h1>;
```

Option 2:

Use ternary expressions instead:

Example

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

```
const x = 5;

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

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.

Example

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>;
}
```

Get Certified!

Complete the React modules, do the exercises, take the exam and become w3schools certified!!

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:

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

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:

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

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

Components in Components

We can refer to components inside other components:

Example

Use the Car component inside the Garage component:

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

function Garage() {
  return (
    <>
      <h1>Who lives in my Garage?</h1>
      <Car />
    </>
  );
}

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

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.

Example

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';
import Car from './Car.js';

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

Continua

https://www.w3schools.com/react/react_class.asp