# 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>
```

# 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>;
```

**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

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>;

}
```

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.

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

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:

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

# 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 />);
```

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 />);
```

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 />);
```

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

**Hello World**

The smallest React example looks like this:

```
ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('root')
);
```

It displays a heading saying “Hello, world!” on the page.

**Introducing JSX**

Consider this variable declaration:

```
const element = <h1>Hello, world!</h1>;
```

This funny tag syntax is neither a string nor HTML.

It is called JSX, and it is a **syntax extension to JavaScript**. We recommend using it with React to describe what the UI should look like. JSX may remind you of a template language, but it comes with the full power of JavaScript.
JSX produces React “elements”. We will explore rendering them to the DOM in the next section. Below, you can find the basics of JSX necessary to get you started.

**Why JSX?**

React embraces the fact that rendering logic is inherently coupled with other UI logic: how events are handled, how the state changes over time, and how the data is prepared for display.

Instead of artificially separating *technologies* by putting markup and logic in separate files, React separates *concerns* with loosely coupled units called "components" that contain both. We will come back to components in a further section, but if you're not yet comfortable putting markup in JS, this talk might convince you otherwise.
React doesn't require using JSX, but most people find it helpful as a visual aid when working with UI inside the JavaScript code. It also allows React to show more useful error and warning messages.

With that out of the way, let's get started!

**Embedding Expressions in JSX**

In the example below, we declare a variable called name and then use it inside JSX by wrapping it in curly braces:

```
const name = 'Josh Perez';const element = <h1>Hello, {name}</h1>;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

You can put any valid JavaScript expression inside the curly braces in JSX. For example, 2 + 2, user.firstName, or formatName(user) are all valid JavaScript expressions.
In the example below, we embed the result of calling a JavaScript function, formatName(user), into an <h1> element.

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};

const element = (
  <h1>
    Hello, {formatName(user)}!  </h1>
);

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

**Try it on CodePen**

We split JSX over multiple lines for readability. While it isn't required, when doing this, we also recommend wrapping it in parentheses to avoid the pitfalls of automatic semicolon insertion.

**JSX is an Expression Too**

After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.
This means that you can use JSX inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions:

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;  }
  return <h1>Hello, Stranger.</h1>;}
```

**Specifying Attributes with JSX**

You may use quotes to specify string literals as attributes:

```
const element = <div tabIndex="0"></div>;
```

You may also use curly braces to embed a JavaScript expression in an attribute:

```
const element = <img src={user.avatarUrl}></img>;
```

Don't put quotes around curly braces when embedding a JavaScript expression in an attribute. You should either use quotes (for string values) or curly braces (for expressions), but not both in the same attribute.

**Warning:**

Since JSX is closer to JavaScript than to HTML, React DOM uses camelCase property naming convention instead of HTML attribute names.
For example, class becomes className in JSX, and tabindex becomes tabIndex.

**Specifying Children with JSX**

If a tag is empty, you may close it immediately with />, like XML:

```
const element = <img src={user.avatarUrl} />;
```

JSX tags may contain children:

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

**JSX Prevents Injection Attacks**

It is safe to embed user input in JSX:

```
const title = response.potentiallyMaliciousInput;
// This is safe:
const element = <h1>{title}</h1>;
```

By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

**JSX Represents Objects**

Babel compiles JSX down to React.createElement() calls.

These two examples are identical:

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

React.createElement() performs a few checks to help you write bug-free code but essentially it creates an object like this:

```
// Note: this structure is simplified
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
};
```

These objects are called "React elements". You can think of them as descriptions of what you want to see on the screen. React reads these objects and uses them to construct the DOM and keep it up to date.

We will explore rendering React elements to the DOM in the next section.

**Tip:**

We recommend using the "Babel" language definition for your editor of choice so that both ES6 and JSX code is properly highlighted.

# React Components

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

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

# Create a Class Component

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

The component has to include the `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>;
  }
}
```

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

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

## Example

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

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

# Create a Function Component

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

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

## Example

Create a Function component called `Car`

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

Once again your React application has a Car component.

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

## Example

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

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

# Component Constructor

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

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

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

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

## Example

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

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

Use the color property in the render() function:

## Example

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

# Props

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

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

## Example

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

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

# Components in Components

We can refer to components inside other components:

## Example

Use the Car component inside the Garage component:

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

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

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

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

# Components in Files

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

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

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

## Example

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

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

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

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

## Example

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

```
import React from 'react';
import ReactDOM from 'react-dom';
import Car from './App.js';

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

Props are arguments passed into React components.

Props are passed to components via HTML attributes.

# React Props

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

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

## Example

Add a "brand" attribute to the Car element:

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

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

## Example

Use the brand attribute in the component:

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

```
  }
}
```

# Pass Data

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

## Example

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

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

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

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

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

## Example

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

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

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

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

Or if it was an object:

## Example

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

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

```
}

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

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

# Props in the Constructor

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

## Example

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

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

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

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

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

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

# Creating the `state` Object

The `state` object is initialized in the constructor:

## Example:

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:

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

# Changing the `state` Object

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

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

## Example:

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

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

```
        <button
          type="button"
          onClick={this.changeColor}
        >Change color</button>
      </div>
    );
  }
}
```

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).

## React Component Life-Cycle

In ReactJS, every component creation process involves various lifecycle methods. These lifecycle methods are termed as component's lifecycle. These lifecycle methods are not very complicated and called at various points during a component's life. The lifecycle of the component is divided into **four phases**. They are:

1. Initial Phase
2. Mounting Phase -  Birth of your component
3. Updating Phase - Growth of your component
4. Unmounting Phase - Death of your component

render()

The *render()* method is the most used lifecycle method. You will see it in all React classes. This is because *render()* is the only required method within a class component in React.

As the name suggests it handles the rendering of your component to the UI. It happens during the **mounting** and **updating** of your component.

Each phase contains some lifecycle methods that are specific to the particular phase. Let us discuss each of these phases one by one.

1. Initial Phase

It is the birth phase of the lifecycle of a ReactJS component. Here, the component starts its journey on a way to the DOM. In this phase, a component contains the default Props and initial State. These default properties are done in the constructor of a component. The initial phase only occurs once and consists of the following methods.

- getDefaultProps()
  It is used to specify the default value of this.props. It is invoked before the creation of the component or any props from the parent is passed into it.

- getInitialState()
  It is used to specify the default value of this.state. It is invoked before the creation of the component.

2. Mounting Phase

In this phase, the instance of a component is created and inserted into the DOM. It consists of the following methods.

- componentWillMount()
  This is invoked immediately before a component gets rendered into the DOM. In the case, when you call setState() inside this method, the component will not re-render.
- componentDidMount()
  This is invoked immediately after a component gets rendered and placed on the DOM. Now, you can do any DOM querying operations.
- render()
  This method is defined in each and every component. It is responsible for returning a single root HTML node element. If you don't want to render anything, you can return a null or false value.

3. Updating Phase

It is the next phase of the lifecycle of a react component. Here, we get new Props and change State. This phase also allows to handle user interaction and provide communication with the components hierarchy. The main aim of this phase is to ensure that the component is displaying the latest version of itself. Unlike the Birth or Death phase, this phase repeats again and again. This phase consists of the following methods.

- componentWillRecieveProps()
  It is invoked when a component receives new props. If you want to update the state in response to prop changes, you should compare this.props and nextProps to perform state transition by using this.setState() method.
- shouldComponentUpdate()
  It is invoked when a component decides any changes/updation to the DOM. It allows you to control the component's behavior of updating itself. If this method returns true, the component will update. Otherwise, the component will skip the updating.
- componentWillUpdate()
  It is invoked just before the component updating occurs. Here, you can't change the component state by invoking this.setState() method. It will not be called, if shouldComponentUpdate() returns false.

- render()
  It is invoked to examine this.props and this.state and return one of the following types: React elements, Arrays and fragments, Booleans or null, String and Number. If shouldComponentUpdate() returns false, the code inside render() will be invoked again to ensure that the component displays itself properly.
- componentDidUpdate()
  It is invoked immediately after the component updating occurs. In this method, you can put any code inside this which you want to execute once the updating occurs. This method is not invoked for the initial render.

4. Unmounting Phase

It is the final phase of the react component lifecycle. It is called when a component instance is destroyed and unmounted from the DOM. This phase contains only one method and is given below.

- componentWillUnmount()
  This method is invoked immediately before a component is destroyed and unmounted permanently. It performs any necessary cleanup related task such as invalidating timers, event listener, canceling network requests, or

cleaning up DOM elements. If a component instance is unmounted, you cannot mount it again.