* **What is React?**

ReactJS is a declarative, efficient, and flexible JavaScript library for building reusable UI components. It is an open-source, component-based front-end library responsible only for the view layer of the application. It was created by Jordan Walke, who was a software engineer at Facebook. It was initially developed and maintained by Facebook and was later used in its products like WhatsApp & Instagram. Facebook developed ReactJS in 2011 in its newsfeed section, but it was released to the public in the month of May 2013.

A ReactJS application is made up of multiple components, each component responsible for outputting a small, reusable piece of HTML code. The components are the heart of all React applications. These Components can be nested with other components to allow complex applications to be built of simple building blocks.

ReactJS uses virtual DOM based mechanism to fill data in HTML DOM. The virtual DOM works fast as it only changes individual DOM elements instead of reloading complete DOM every time.

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:

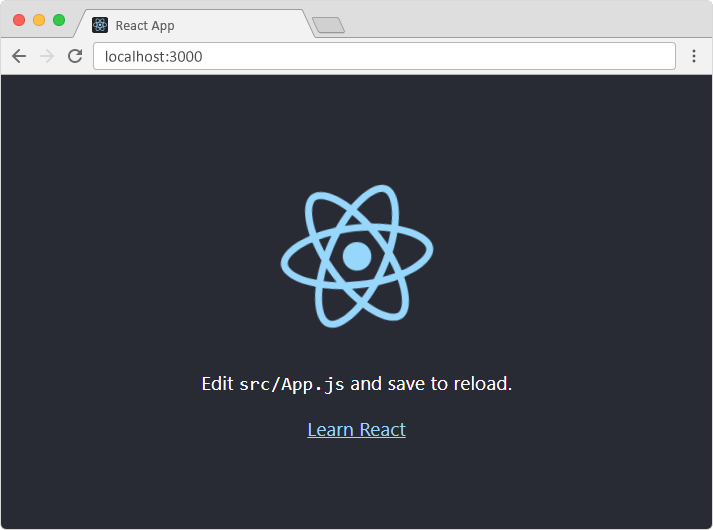
cd my-react-app

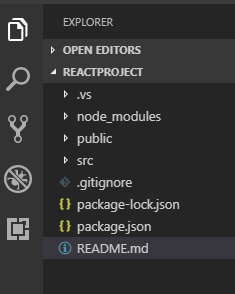
Run this command to run the React application my-react-app:

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.

The result:





In React application, there are several files and folders in the root directory. Some of them are as follows:

1. **node\_modules:** It contains the React library and any other third party libraries needed.
2. **public:** It holds the public assets of the application. It contains the index.html where React will mount the application by default on the <div id="root"></div> element.
3. **src:** It contains the App.css, App.js, App.test.js, index.css, index.js, and serviceWorker.js files. Here, the App.js file always responsible for displaying the output screen in React.
4. **package-lock.json:** It is generated automatically for any operations where npm package modifies either the node\_modules tree or package.json. It cannot be published. It will be ignored if it finds any other place rather than the top-level package.
5. **package.json:** It holds various metadata required for the project. It gives information to npm, which allows to identify the project as well as handle the project?s dependencies.
6. **README.md:** It provides the documentation to read about React topics.

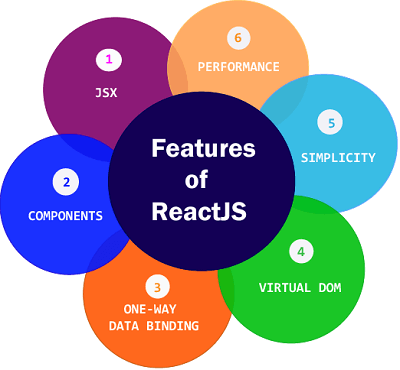
### **React Environment Setup**

Now, open the **src >> App.js** file and make changes which you want to display on the screen. After making desired changes, **save** the file. As soon as we save the file, Webpack recompiles the code, and the page will refresh automatically, and changes are reflected on the browser screen. Now, we can create as many components as we want, import the newly created component inside the **App.js** file and that file will be included in our main **index.html** file after compiling by Webpack.

Next, if we want to make the project for the production mode, type the following command. This command will generate the production build, which is best optimized.

1. $ npm build

* **React Features**



The important features of ReactJS are as following.

* JSX
* Components
* One-way Data Binding
* Virtual DOM
* Simplicity
* Performance
  1. **JSX**

JSX stands for JavaScript XML. It is a JavaScript syntax extension. Its an XML or HTML like syntax used by ReactJS. This syntax is processed into JavaScript calls of React Framework. It extends the ES6 so that HTML like text can co-exist with JavaScript react code. It is not necessary to use JSX, but it is recommended to use in ReactJS.

* 1. **Components**

ReactJS is all about components. ReactJS application is made up of multiple components, and each component has its own logic and controls. These components can be reusable which help you to maintain the code when working on larger scale projects.

* 1. **One-way Data Binding**

ReactJS is designed in such a manner that follows unidirectional data flow or one-way data binding. The benefits of one-way data binding give you better control throughout the application. If the data flow is in another direction, then it requires additional features. It is because components are supposed to be immutable and the data within them cannot be changed. Flux is a pattern that helps to keep your data unidirectional. This makes the application more flexible that leads to increase efficiency.

* 1. **Virtual DOM**

A virtual DOM object is a representation of the original DOM object. It works like a one-way data binding. Whenever any modifications happen in the web application, the entire UI is re-rendered in virtual DOM representation. Then it checks the difference between the previous DOM representation and new DOM. Once it has done, the real DOM will update only the things that have actually changed. This makes the application faster, and there is no wastage of memory.

* 1. **Simplicity**

ReactJS uses JSX file which makes the application simple and to code as well as understand. We know that ReactJS is a component-based approach which makes the code reusable as your need. This makes it simple to use and learn.

* 1. **Performance**

ReactJS is known to be a great performer. This feature makes it much better than other frameworks out there today. The reason behind this is that it manages a virtual DOM. The DOM is a cross-platform and programming API which deals with HTML, XML or XHTML. The DOM exists entirely in memory. Due to this, when we create a component, we did not write directly to the DOM. Instead, we are writing virtual components that will turn into the DOM leading to smoother and faster performance.

## 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.

1. **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 Employee {

constructor(name) {

this.empName = name;

}

}

create objects using the Car class:

Example

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

const e1 = new Employee("xyz");

Exmaple

<!DOCTYPE html>

<html>

<body>

<script>

class Employee {

constructor(name) {

this.empName = name;

}

}

const e1 = new Employee("xyz");

document.write(e1.empName);

</script>

</body>

</html>

**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();

## Arrow Functions

Arrow functions allow us to write shorter function syntax:

hello = function() {

return "Hello World!";

}

### **With Arrow Function:**

hello = () => {

return "Hello World!";

}

### **Arrow Functions Return Value by Default:**

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

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 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.

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);

1. **Variables**

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

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

### **var**

var x = 5.6;

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

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

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

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

### **let**

let x = 5.6;

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

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

let has a *block* scope.

### **const**

const x = 5.6;

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

const has a *block* scope.

## 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:

import React from 'react';

import ReactDOM from 'react-dom/client';

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

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

ReactDOM.render(myList, document.getElementById('root'));

apple

banana

orange

## Destructuring

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

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

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

// old way

const car = vehicles[0];

const truck = vehicles[1];

const suv = vehicles[2];

or

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

const [car, truck, suv] = vehicles;

## Destructuring Objects

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 + '.';

}

With Destructuring

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 + '.';

}

## 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.

<script>

const numbersOne = [1, 2, 3];

const numbersTwo = [4, 5, 6];

const numbersCombined = [...numbersOne, ...numbersTwo];

document.write(numbersCombined);

</script>

Output

1,2,3,4,5,6

## Modules

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

This makes it easier to maintain the code-base.

ES Modules rely on the import and export statements.

Export

You can export a function or variable from any file.

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

There are two types of exports: Named and Default.

## Named Exports

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

### **In-line individually:**

person.js

export const name = "Jesse"

export const age = 40

### **All at once at the bottom:**

person.js

const name = "Jesse"

const age = 40

export { name, age }

## Default Exports

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

You can only have one default export in a file.

### **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";

## 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:

<!DOCTYPE html>

<html>

<body>

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

<script>

function renderApp(){

document.getElementById("demo").innerHTML = "Welcome!";

}

function renderLogin()

{

document.getElementById("demo").innerHTML = "Please log in";

}

let authenticated = true;

authenticated ? renderApp() : renderLogin();

</script>

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

</body>

</html>

# **React Render HTML**

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.

import React from 'react';

import ReactDOM from 'react-dom/client';

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

## 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.

import React from 'react';

import ReactDOM from 'react-dom/client';

const myelement = (

<table>

<tr>

<th>Name</th>

</tr>

<tr>

<td>John</td>

</tr>

<tr>

<td>Elsa</td>

</tr>

</table>

);

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

## The Root Node

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

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

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

### **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 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.

## 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>;

}

## 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:

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car />);

## Props

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

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

You will learn more about props in the next chapter.

### **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>;

}

const root = ReactDOM.createRoot(document.getElementById('root'));

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

## 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 />

</>

);

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Garage />);

# **React Class Components**

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

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

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

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

## 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.

### **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:

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car />);

## Component Constructor

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

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

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

You will learn more about state later in this tutorial.

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

### **Example**

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

class Car extends React.Component {

constructor() {

super();

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

}

render() {

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

}

}

Use the color property in the render() function:

### **Example**

class Car extends React.Component {

constructor() {

super();

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

}

render() {

return <h2>I am a {this.state.color} Car!</h2>;

}

}

## Props

Another way of handling component properties is by using props.

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

You will learn more about props in the next chapter.

### **Example**

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

class Car extends React.Component {

render() {

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

}

}

const root = ReactDOM.createRoot(document.getElementById('root'));

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

## 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>;

}

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car model="Mustang"/>);

## 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>

);

}

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Garage />);

## Components in Files

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

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

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

### **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.

import React from 'react';

import ReactDOM from 'react-dom/client';

import Car from './Car.js';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car />);

## React Class 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:

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>

);

}

}

[Run Example »](https://www.w3schools.com/react/showreact.asp?filename=demo2_react_state)

## Changing the state Object

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

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

### **Example:**

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

class Car extends React.Component {

constructor(props) {

super(props);

this.state = {

brand: "Ford",

model: "Mustang",

color: "red",

year: 1964

};

}

changeColor = () => {

this.setState({color: "blue"});

}

render() {

return (

<div>

<h1>My {this.state.brand}</h1>

<p>

It is a {this.state.color}

{this.state.model}

from {this.state.year}.

</p>

<button

type="button"

onClick={this.changeColor}

>Change color</button>

</div>

);

}

}

# **React Props**

Props are arguments passed into React components.

Props are passed to components via HTML attributes.

props stands for properties.

## React Props

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

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

### **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.