**React component guide: Class vs functional -** Leonard Kong May 24, 2022

If you’ve been coding in JavaScript for a while, you’re likely familiar with **React** (or ReactJS), the front-end library for building user interfaces. You probably know that it’s the **most popular front-end framework in the tech world by usage**[[1]](https://2021.stateofjs.com/en-US/libraries/front-end-frameworks/) thanks to its flexibility, widespread use cases, ease of use, and broad support.

Today, we will go deeper into React and explore one of its fundamental building blocks: components. Using code examples, we’ll look at what React components do, compare **class and functional components**, and discuss which type you should use.

**We’ll cover**:

* [What is a React component?](https://www.educative.io/blog/react-component-class-vs-functional?eid=5082902844932096#react-component)
* [What is a class component?](https://www.educative.io/blog/react-component-class-vs-functional?eid=5082902844932096#class-component)
* [What is a functional component?](https://www.educative.io/blog/react-component-class-vs-functional?eid=5082902844932096#functional-component)
* [Comparing class and functional components](https://www.educative.io/blog/react-component-class-vs-functional?eid=5082902844932096#comparing)
* [Which component should you use?](https://www.educative.io/blog/react-component-class-vs-functional?eid=5082902844932096#which-to-use)
* [Wrapping up and next steps](https://www.educative.io/blog/react-component-class-vs-functional?eid=5082902844932096#next-steps)

**What is a React component?**

Like all programming languages, JavaScript supports **functions** allowing reusable pieces of business logic to be inserted into larger chunks of code. This hides complexity and illustrates the concept of a **subroutine**.

**React components** serve the same purpose, except React components effectively divide the UI into reusable components that return HTML. In a sense, React components are subroutines for user interfaces.

**Note**: React components are independent mixtures of JavaScript code and HTML fragments returning markup (HTML). The JavaScript provides the business logic and enables the creation of the correct HTML to be inserted into the DOM. (CSS provides the styling.)

We can illustrate this with some code. You can easily create a JS file that returns an HTML fragment:

1

2

3

return (

    <p>Hello, World</p>

)





It’s more complicated to get the JS file to return HTML defined elsewhere. One way of doing that is to use an HTML markup placeholder:

1

2

3

return (

    <WelcomeMessage />

)





The <WelcomeMessage /> markup is a placeholder for what a **JSX file** defines with the requisite HTML.

**Definition**: JSX stands for JavaScript XML and allows us to write HTML in React.

Here’s how that would look:

1

2

3

4

5

6

function WelcomeMessage() {

  return (

    <p>Hello, World</p>

  )

}

export default WelcomeMessage;





Here, we’ve defined a function named WelcomeMessage() that can be rendered in place of the placeholder defined in the main JS file. You just need to import the function into the main JS file like this:

1

2

3

4

5

import WelcomeMessage from "./WelcomeMessageComponent";

return (

<WelcomeMessage />

)





As a result, the JS file will return <p>Hello, World</p> instead of the placeholder <WelcomeMessage />.

In our example, we have created a React component that would return the element <p>Hello, World</p>. But React components are not limited to static HTML. You can add JavaScript logic to make them more usable.

Let’s make this example a bit more interesting. We can replace the word “World” in the HTML with a variable {place}. We can further add JavaScript logic to assign a value to our variable based on a condition:

1

2

3

4

5

6

7

8

9

10

11

12

let place = "Unknown";

switch (SomeCondition)

{

  case condition1: place = "Ukraine"; break;

  case condition2: place = "Russia"; break;

  case condition3: place = "US"; break;

  default: place = "World"; break;

}

return (

  <p>Hello, {place}</p>

)





Here, React will replace {place} with text, based on a condition in the WelcomeMessage component. The WelcomeMessage component will return an HTML element with textual value substituted for {place}.

This is the **essence of a React component**: to enable business logic to modify HTML fragments and return plain HTML for insertion into the DOM.

React has two ways of creating components, class, and functional components. Despite syntactical differences, they both return a piece of HTML created from a mixture of JavaScript and HTML. Let’s take a closer look at both types of components.

**What is a class component?**

A **class component** is an ECMAScript 6 (ES6) JavaScript class that you define in code like this:

1

2

3

4

5

6

7

8

9

10

11

12

import { Component } from 'react';

class MyComponent extends React.Component {

  constructor(props) {

    super(props);

    this.state = { currState: true }

  }

  render() {

    <div>

      <p>Hello, World!</p>

    </div>

  }

}





Class components extend from the **React.Component class**. React.Component objects have **state**, meaning the object can hold information that can change over the lifetime of the object. They can also respond to [**lifecycle methods**](https://www.educative.io/blog/the-evolution-of-react#lifecycle-methods), like ComponentDidMount(), ComponentDidUpdate(), and ComponentWillUnMount().

**Note**: Lifecycle methods enable updated state information to trigger a re-render, which updates the DOM with revised HTML markup.

Class components can also have properties (also known as **props**) passed to them. Props are passed as arguments to the constructor and should be passed to the parent component class by calling super(props). Props are accessible throughout the life of the object.

The render() method is required in class components and used to return an HTML element.

**What is a functional component?**

**Functional components** are JavaScript functions. There are two ways of creating them. The first is by using the function keyword:

1

2

3

4

5

6

7

8

function MyComponent(props) {

  return (

    <div>

      <p>Hello, World</p>

      <p>Have a nice day!</p>

    </div>

  );

}





You can also use the arrow function syntax to create functional components:

1

2

3

4

5

6

7

8

const MyComponent = (props) => {

  return (

    <div>

      <p>Hello, World</p>

      <p>Have a nice day!</p>

    </div>

  );

}





Functional components return a single HTML element. To return more elements, you can wrap them in a topmost <div> element.

React has a built-in element for doing the same thing, the **React.Fragment**. Here’s what it looks like:

1

2

3

4

5

6

7

8

const MyComponent = (props) => {

  return (

    <React.Fragment>

      <p>Hello, World</p>

      <p>Have a nice day!</p>

    </React.Fragment>

  );

}





React treats the React.Fragment like a single HTML element (e.g., <div>), but doesn’t render it, so it doesn’t contribute to the DOM.

Although functional components have the word “component” in their name, they don’t extend from the React.Component object. Functional components are just JavaScript functions that use [**React hooks**](https://www.educative.io/blog/react-hooks-tutorial-todo-list) to provide the equivalent functionality as class components. We call them components because the functions are constructed with single props object arguments and return React elements.

A JavaScript function name can start with a lowercase letter, though this isn’t required. On the other hand, functional component names must begin with an uppercase letter.

Since functional components are not objects, you must use [React hooks](https://www.educative.io/blog/react-useeffect-hook) to manage state and lifecycle events. Hooks were not introduced into React until version 16.8, but they have effectively made class components redundant.

Functional components are also known as stateless components because they aren’t class objects and don’t maintain state. You can pass props to functional components as arguments, and they can return HTML.

**Comparing class and functional components**

Let’s review the differences between class and functional components.

**Class components**:

* Extend from React.Component
* Are also known as stateful components
* Can respond to lifecycle events
* Maintain state information
* Support props
* Require a constructor to store state before they can be used to pass props to the parent class
* Require a render function that returns an HTML element

**Functional components**:

* Don’t extend from React.Component
* Are known as stateless components
* Don’t respond to lifecycle events
* Don’t maintain state information
* Will accept any type of data (props)
* Don’t support a constructor
* Return HTML elements or nothing
* Support React 16.8 hooks

Graphical user interface, application

Description automatically generated

Cheat sheet: React components

**Which component should you use?**

With React version 16.8, **class components have taken a back seat to functional components**. Functional components are more concise, leading to cleaner, less complex code. They don’t include lifecycle methods or inherited members that may or may not be required for code functionality.

Anything that can be done using class components can also be done using functional components. The only exception is that React supports a special class component called **Error Boundaries** that can’t be duplicated as a function component.

Because class components extend from React.Component, they have state and lifecycle methods associated with them. Their presence requires you to better understand when lifecycle events will occur and how to respond to them to manage state. Classes also require additional setup for the component or to make API calls for data, primarily implemented via the constructor. Sharing logic between multiple class objects without implementing design patterns is more difficult, leading to code that is more complex and difficult to maintain.

Deciding which component to use always devolves into a discussion of [**inheritance vs composition**](https://www.educative.io/edpresso/what-is-the-difference-between-composition-and-inheritance). Using functional components encourages composition, while class components lend themselves to inheritance design patterns. Currently, composition is considered a best practice in programming, which is why most new React code uses functional components instead of class components. That said, React still supports class components for legacy purposes.

**Wrapping up and next steps**

In this article, we’ve taken a close look at React components. By now, you should have a good understanding of the role components play in React apps, and the key differences between class and functional components.

From here, you might look into topics like the react-dom package (also known as ReactDOM), which lets you access and modify the DOM. You could also learn about component lifecycles, lifecycle methods, and their parameters. Or you could study other React APIs (like setState), the difference between parent and child components, or writing event handlers.

However you want to advance your skills, you don’t need to rely on the React docs or projects on GitHub for help. To help you master this JavaScript library, we’ve created the learning path [**React Development for Professionals**](https://www.educative.io/path/react-development-for-professionals). Designed for those already familiar with React, this path covers hooks, design patterns, and thinking more abstractly about creating applications.