React JS

1. Declarative

* React makes it painless to create interactive UIs.
* Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.

1. Component-Based

* Build encapsulated components that manage their own state, then compose them to make complex UIs.

**Points to remember:**

function (longName)

{

return { longName };

}

is equivalent to

function (longName)

{

return { longName: longName };

}

use - spread operator

const stringList = ["Mohan","Fasale","Robert"];

const name = "Priya";

const result = [...stringList, name];

console.log(stringList); // ["Mohan","Fasale","Robert"]

console.log(result); // ["Mohan","Fasale","Robert","Priya"]

We sometimes bind variables using destructuring assignment, usually with shorthand property names.

For example,

var { longPropName } = props;

is equivalent to

var longPropName = props.longPropName;

arrow function don't have a context of this.

a simple answer for that is arrow don't know about 'this' keyword it simple checks up the hierarchical order to get 'this'

**Introducing JSX**

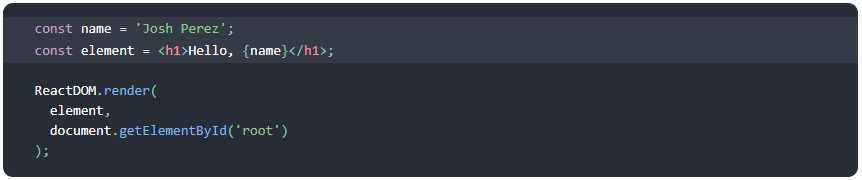


This funny tag syntax is neither a string nor HTML.

It is called JSX, and it is a syntax extension to JavaScript.

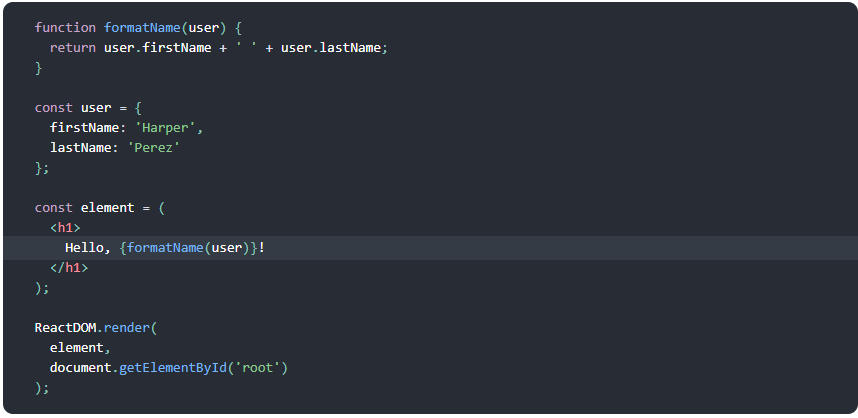
**Expressions in JSX**

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



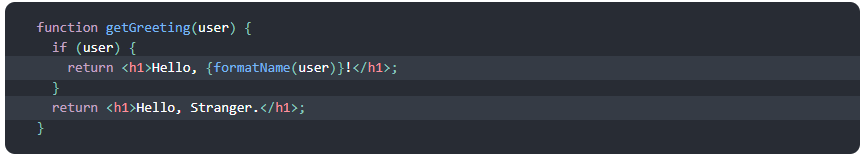
You can put any valid JavaScript expression inside the curly braces in JSX. For example, 2 + 2, user.firstName, or formatName(user) are all valid JavaScript expressions.

In the example below, we embed the result of calling a JavaScript function, formatName(user), into an <h1> element.



**JSX is an Expression Too**

you can use JSX inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions:



**Specifying Attributes with JSX**

You may use quotes to specify string literals as attributes:



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



**Specifying Children with JSX**

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



JSX tags may contain children:



**Rendering into the DOM**

Applications built with React usually have a root DOM node

To render a React element into a root DOM node, pass both root node and element to ReactDOM.render():

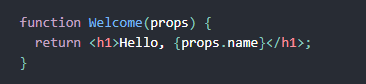


**Components and Props**

Components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen

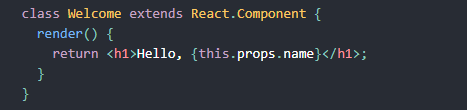
**Function and Class Components**

The simplest way to define a component is to write a JavaScript function:



This function is a valid React component because it accepts a single “props” (which stands for properties) object argument with data and returns a React element. We call such components “function components” because they are literally JavaScript functions.

You can also use an ES6 class to define a component:



**Components**

**React Elements**



**user-defined components**



For example, this code renders “Hello, Sara” on the page:



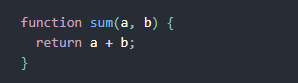
1. We call ReactDOM.render() with the <Welcome name="Sara" /> element.
2. React calls the Welcome component with {name: 'Sara'} as the props.
3. Our Welcome component returns a <h1>Hello, Sara</h1> element as the result.
4. React DOM efficiently updates the DOM to match <h1>Hello, Sara</h1>.

**Note: Always start component names with a capital letter.**

React treats components starting with lowercase letters as DOM tags. For example, <div /> represents an HTML div tag, but <Welcome /> represents a component and requires Welcome to be in scope.

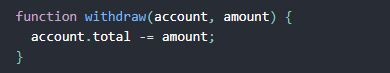
**All React components must act like pure functions with respect to their props.**

Whether you declare a component as a function or a class, it must never modify its own props. Consider this sum function:



Such functions are called “pure” because they do not attempt to change their inputs, and always return the same result for the same inputs.

In contrast, this function is impure because it changes its own input:



**State and Lifecycle**

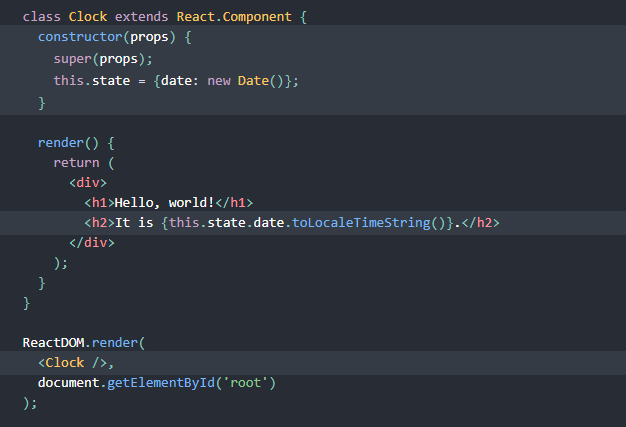
State is similar to props, but it is private and fully controlled by the component

In this section, we will learn how to make the **Clock** component

It will set up its own timer

and update itself every second.

**Clock Component:**



Next, we’ll make the Clock set up its own timer and update itself every second.

## Adding Lifecycle Methods to a Class

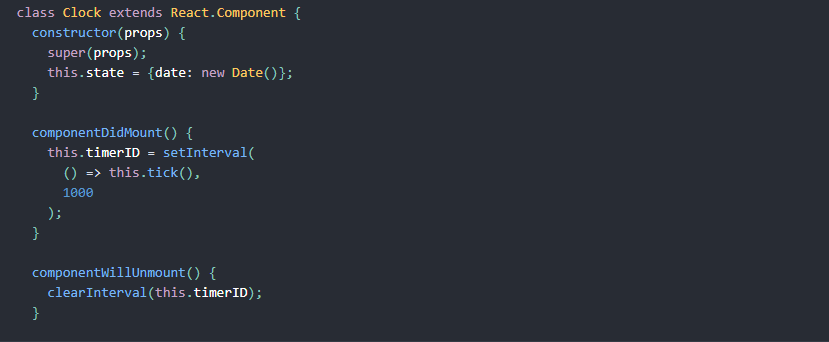
In applications with many components, it’s very important to free up resources taken by the components when they are destroyed.

We want to [set up a timer](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/setInterval) whenever the Clock is rendered to the DOM for the first time. This is called “mounting” in React.

We also want to [clear that timer](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/clearInterval) whenever the DOM produced by the Clock is removed. This is called “un-mounting” in React.

These methods are called “**lifecycle methods**”.

We can declare special methods on the component class to run some code when a component mounts and unmounts:





Now the clock ticks every second.

Let’s quickly recap what’s going on and the order in which the methods are called:

1. When <Clock /> is passed to ReactDOM.render(), React calls the constructor of the Clock component. Since Clock needs to display the current time, it initializes this.state with an object including the current time. We will later update this state.
2. React then calls the Clock component’s render() method. This is how React learns what should be displayed on the screen. React then updates the DOM to match the Clock’s render output.
3. When the Clock output is inserted in the DOM, React calls the componentDidMount() lifecycle method. Inside it, the Clock component asks the browser to set up a timer to call the component’s tick() method once a second.
4. Every second the browser calls the tick() method. Inside it, the Clock component schedules a UI update by calling setState() with an object containing the current time. Thanks to the setState() call, React knows the state has changed, and calls the render() method again to learn what should be on the screen. This time, this.state.date in the render() method will be different, and so the render output will include the updated time. React updates the DOM accordingly.
5. If the Clock component is ever removed from the DOM, React calls the componentWillUnmount() lifecycle method so the timer is stopped.

**Using State Correctly**

There are three things you should know about setState().

1. **Do Not Modify State Directly**



Instead, use setState():



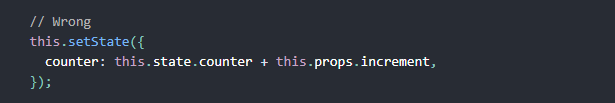
The only place where you can assign this.state is the constructor.

1. **State Updates May Be Asynchronous**

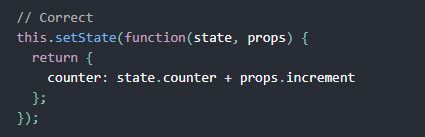
React may batch multiple setState() calls into a single update for performance.

Because this.props and this.state may be updated asynchronously, you should not rely on their values for calculating the next state.

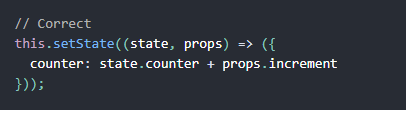
For example, this code may fail to update the counter:



To fix it, use a second form of setState() that accepts a function rather than an object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:



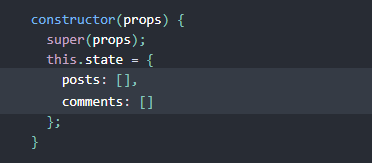
We can also use arrow function as :



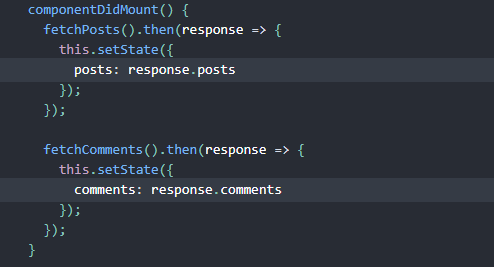
1. **State Updates are Merged**

When you call setState(), React merges the object you provide into the current state.

For example, your state may contain several independent variables:



Then you can update them independently with separate setState() calls:



The merging is shallow, so this.setState({comments}) leaves this.state.posts intact, but completely replaces this.state.comments.

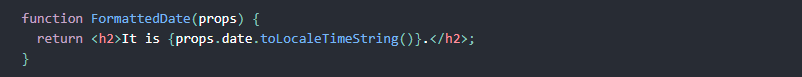
**The Data Flows Down**

State is not accessible to any other component than the one that owns and sets it

A component may choose to pass its state down as props to its child components:



The FormattedDate component would receive the date in its props

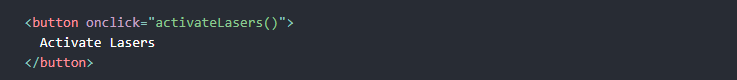


This is commonly called a “top-down” or “unidirectional” data flow.

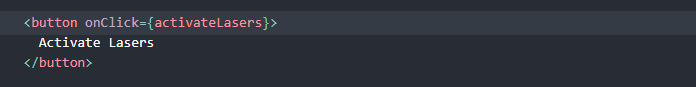
**Handling Events**

With JSX you pass a function as the event handler, rather than a string.

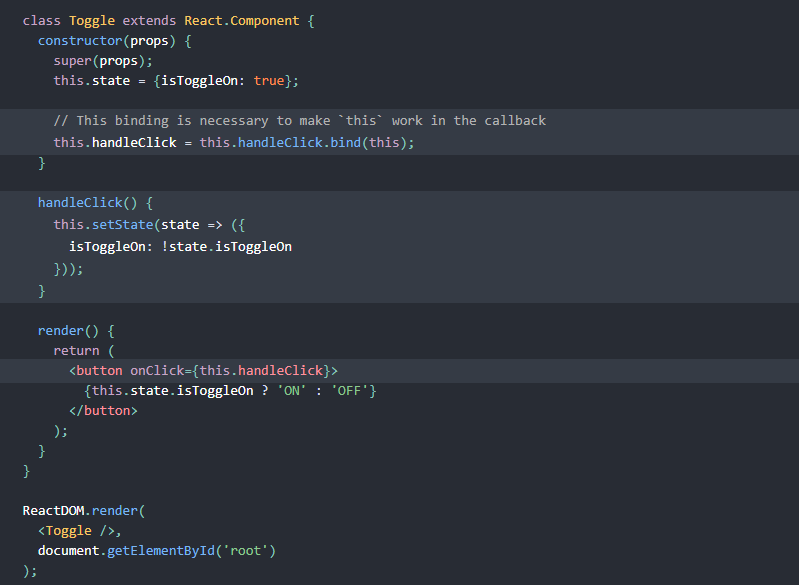
For example, the HTML:



is slightly different in React:



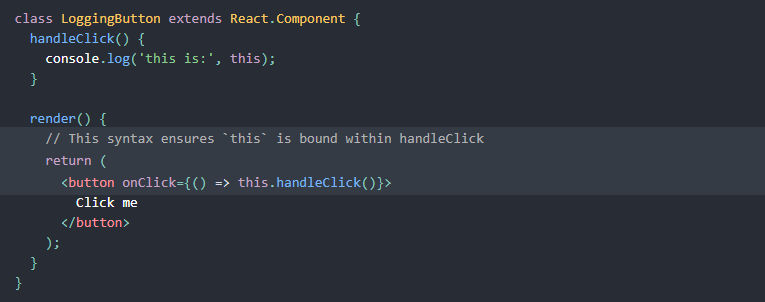
example



If you forget to bind this.handleClick and pass it to onClick, this will be undefined when the function is actually called.

This is not React-specific behavior; it is a part of how functions work in JavaScript. Generally, if you refer to a method without () after it, such as onClick={this.handleClick}, you should bind that method.

If calling bind annoys you , you can use an arrow function in the callback



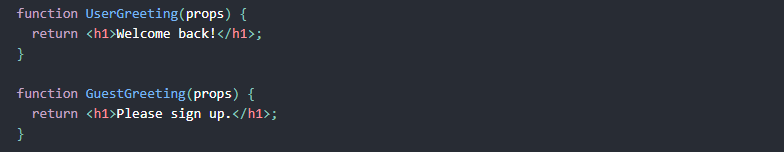
**Conditional Rendering**

In React, you can create distinct components that encapsulate behaviour you need. Then, you can render only some of them, depending on the state of your application.

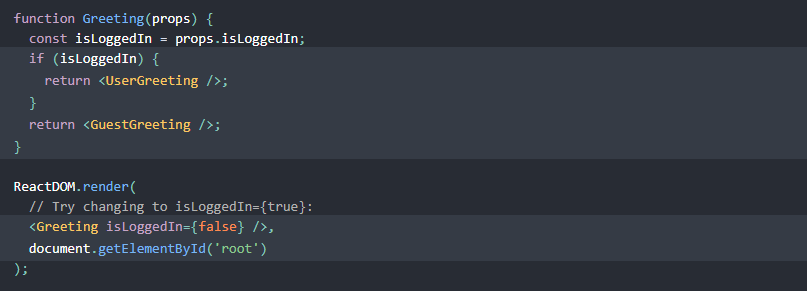
Conditional rendering in React works the same way conditions work in JavaScript.

Use JavaScript operators like if or the conditional operator to create elements representing the current state, and let React update the UI to match them.

Consider these two components:



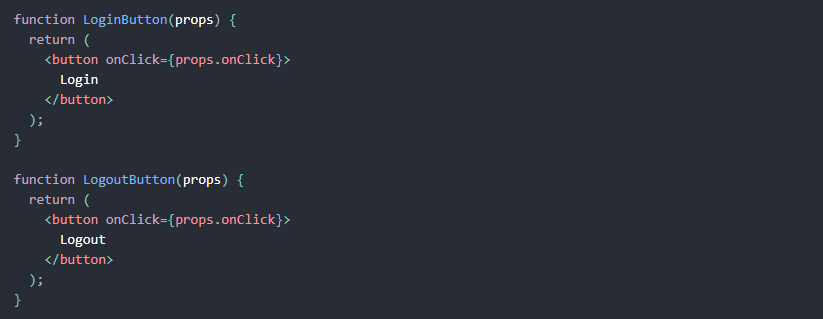
We’ll create a Greeting component that displays either of these components depending on whether a user is logged in:



**Element Variables**

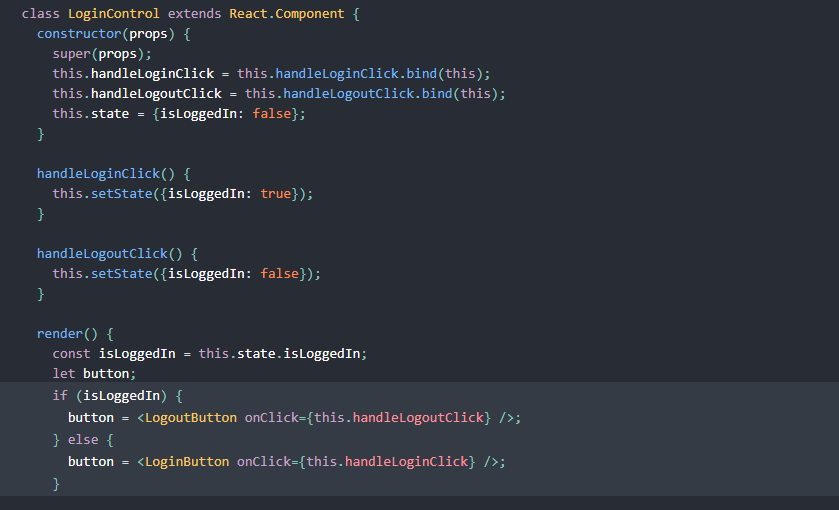
You can use variables to store elements.

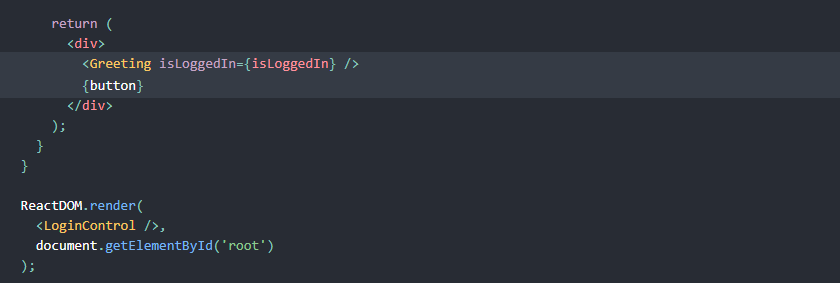
Consider these two new components representing Logout and Login buttons:



In the example below, we will create a component called LoginControl.

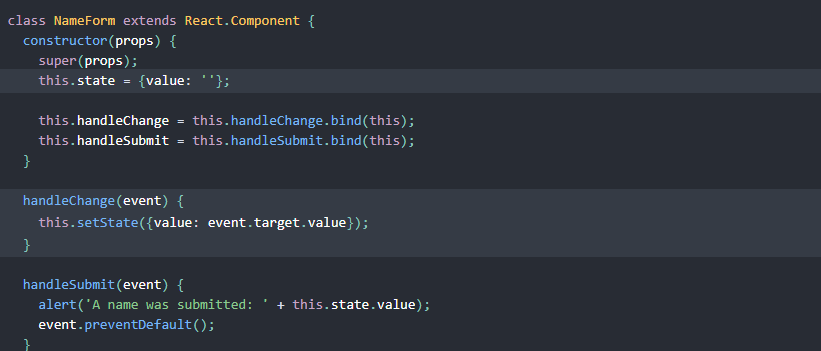
It will render either <LoginButton /> or <LogoutButton /> depending on its current state.

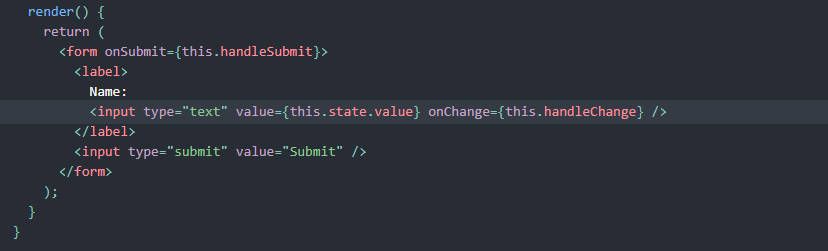




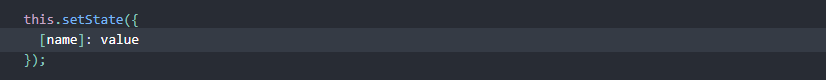
Here *BUTTON is a variable and stores element .*

**Forms**





## Handling Multiple Inputs



**Composition**

React has a powerful composition model, and we recommend using composition instead of inheritance to reuse code between components.

Some components don’t know their children ahead of time. This is especially common for components like Sidebar or Dialog that represent generic “boxes”.

We recommend that such components use the special children prop to pass children elements directly into their output:

