

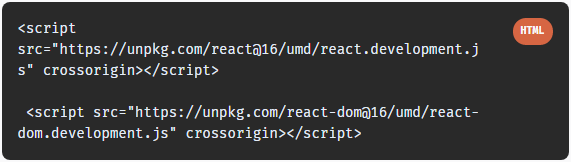
# 1. Getting started

# What is Front-End Development?

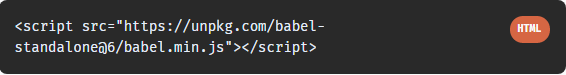
**Front-end development** refers to what the end user (also commonly referred to as the "**client**") can see. In the most basic forms, front-end development consists of HTML, CSS, and JavaScript.  
  
As a developer, you will find that it is very easy for your front-end (website, web application, etc.) to become very complex and have a lot of different moving parts. It makes solving problems much more difficult when you have to go through a maze of code to find the issue.  
  
Eventually, developers decided that there must be a better way to manage all of that code, so they created libraries that could make life easier. **React**was one of those libraries.

# Adding React

React can be added to a website without any special tools and installations.  
  
First, we need to add the React library as two **script** tags to the **head** of our HTML document:

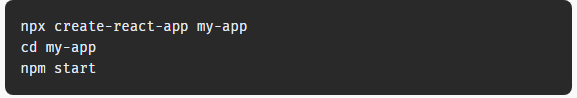


Next, we need to add another script, to enable the use of **JSX**.  
JSX is a syntax extension to JavaScript, and it is recommended to be used with React.  
Don't worry, we will learn more about JSX in the next lessons. For now, let's just add the following script tag:

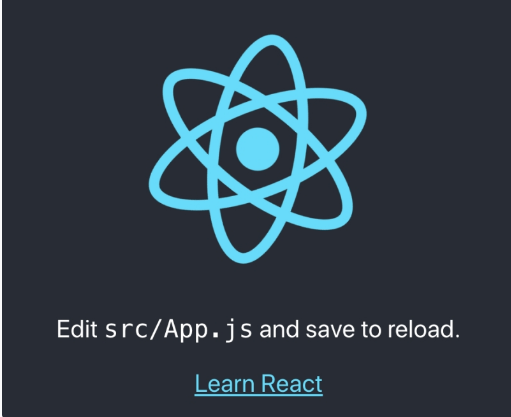


# 2. Create React App

In the previous lesson we learned how to add React to a simple HTML document using the script tags.  
However, real web apps have a different scale, contain multiple files, use 3rd party libraries, etc.  
  
Facebook has created a handy tool called [**Create React App**](https://create-react-app.dev/) that makes it easy to setup a React project with just a simple command!  
  
To get started, make sure you have a recent version of [**Node**](https://nodejs.org/en/) installed on your machine.  
Run the following commands in the Terminal to create and start a React app called "**my-app**":

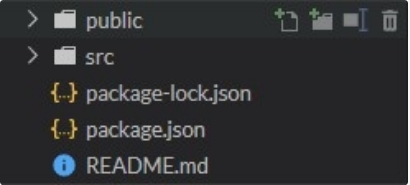


This will install all the required dependencies, configure and start the project on **localhost:3000**.  
  
This is the default output of our project in the browser:

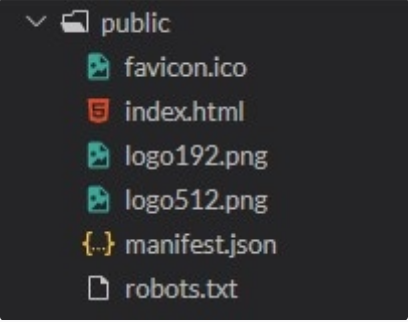


# 3. Project Structure

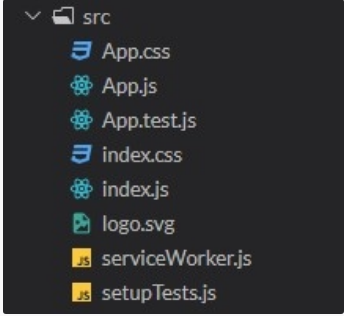
# Let's explore the structure of our project by opening it using a code editor. We will be using **Visual Studio Code** in our example, but you are free to use any code editor.



The **public** folder contains files related to how the application will display on the client, the most important of those being **index.html**, which is the HTML template of our page:



The **src** folder contains all of the JavaScript, CSS, and image files that will be compiled into a bundle file and injected into **index.html**



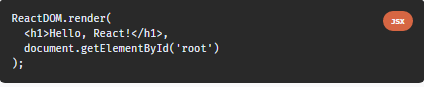
How is React compiled into a bundle file? It uses what is called a "file loader". In the case of Create React App, **Webpack** is used.  
Webpack creates a "bundle" file containing the content of multiple files that need to be "bundled" together and it is all added together into a single file. Instead of making the HTML file go and find multiple files, which can slow down load times tremendously, it only has to find one file.



While there are other files in the **src**folder that come with **Create React App** when it is generated, the two files below are the only critical files:  
• **index.js**: This file is the entry point into our application. In our code, a method called **ReactDOM.render()** is used to find an element with**id="root"** in the HTML and add our React application inside of that element (similar to the previous lesson).  
• **App.js**: This file is the main **component** that will be rendered to the DOM, which currently includes the React logo image and the default text, that we see in the output.

# Changing the Output

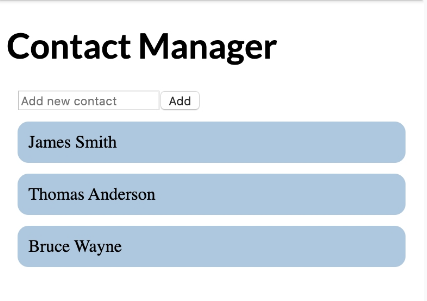
Now, when we know how to create and run a React project, let's change the default output to a simple Hello message.  
To do that, we need to open **src/index.js** and change the code to the following:





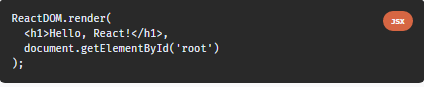
# 4. Course Project

Throughout this course, we'll help you practice and create your own **Contact Manager** app using React, so you'll retain what you've learned and be able to put it to use.  
  
Our Contact Manager will allow to view the list of contacts and add new ones to the list.

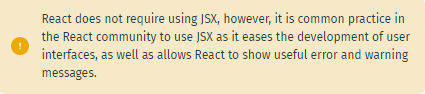


# 5.

# What is JSX? In the previous module we used the following code to show an output using React:



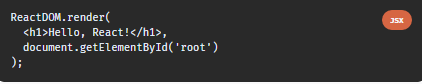
Let's start to break down the code and understand each part of it.  
We will start with the **<h1>Hello, React! </h1>** element.  
  
As you can see, the element is not in quotes to represent a string. It's like an HTML element, however we use it right in the JavaScript code!  
This is called **JSX**, and it is a syntax extension to JavaScript. It allows us to build UI elements right in the JavaScript code!



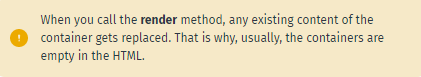
**25 Comments**

# Intro to

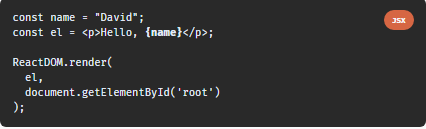
# Let's have a look at our code again:



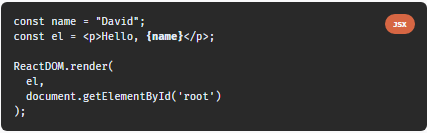
The code calls React's **render** method, and passes it two arguments, a JSX element and a container. The **render** method displays the provided element in the container, which, in our case, is the HTML element with **id="root"**.



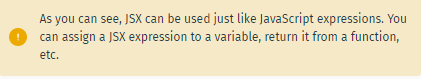
# Expressions in JSX we can use any JavaScript expression inside JSX using curly braces.

  
In the

For example above, we use the



As you can see, JSX can be used just like JavaScript expressions. You can assign a jsx expression to variable **name** in the JSX element, return it from a function, etc.



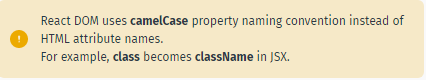
# Attributes in JSX

We can specify attributes using quotes, just like in HTML:

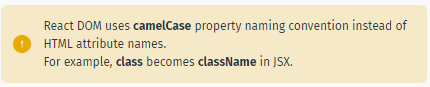




When using a JavaScript expression as the attributes value, the quotes should not be used:

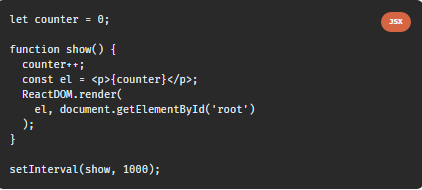


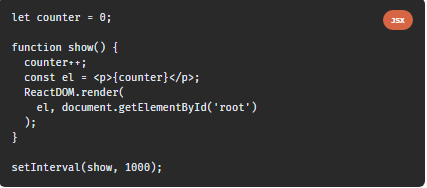




# How Does JSX Work?

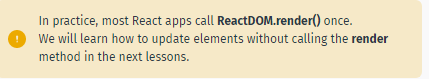
When the JSX expressionsjsx expression are compiledcomplied, they are converted into JavaScript  **objects**, representing React elements.  
 React then uses these elements to build the corresponding HTML DOM and display it in the browser.  
Let's

Let’s create a counter app, that increments a counter variable every second and displays it on the page as a paragraph:  


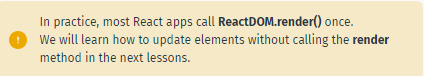


We use can **setInterval** to call the  **show**  function every second and render the counter element on the page.

One of the great features of React is that it updates only the elements that need an update. You can inspect the HTML output of the example above and see that only the text in the paragraph gets updated every second.

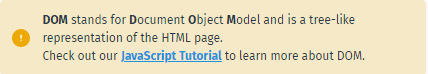


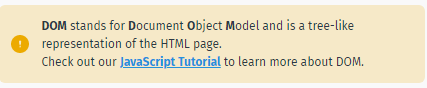
6.



We learned in the previous part that React updates only the elements that are necessary.  
This allows React apps to be much faster than apps built with other front-end technologies.

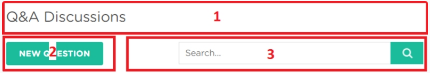
But how does React achieve that?  
React uses a **Virtual DOM**, which is a lightweight representation of the DOM.  
When an element gets changed, it is first updated in the Virtual DOM. That process is fast, as the virtual DOM is represented by simple objects.   
  
After that, React compares the Virtual DOM to its previous state and only applies the DOM updates necessary to bring the DOM to the desired state.

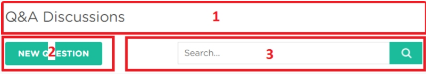




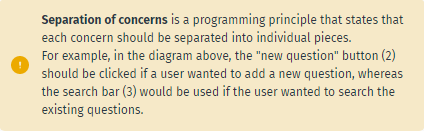
# 7. **Components**

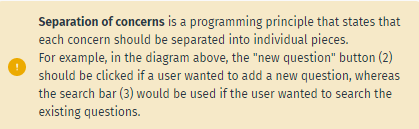
Components let you split the page into independent and reusable parts. Let’sLet's visualize this by taking a look at a piece of the SoloLearn Discussions page:





Notice that the page can be split into multiple parts. Each of these "parts" are a **component**.  
The heading is a component, the "new question" button is a component, and the search bar is its own component.   
This makes organizing the page leaps and bounds easier, but even more importantly, components allow us as the developers to separate concerns from one another.

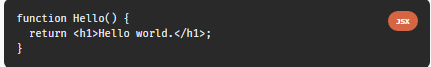


In

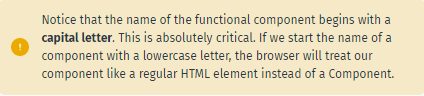
Functional Components  
in React, there are two types of components that you can use: **Functional Components**and**Class Components**.  
In this part, we will talk about functional components. A functional component is a simple JavaScript function:

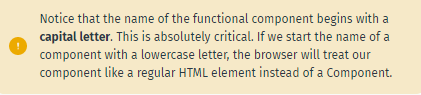
# Functional Components

A functional component is a simple JavaScript function: **Hello (){ return <h1>hello world.</h1>; }**



The code above defined a functional component called **Hello** that returns a simple React element.



. 

# Rendering Components

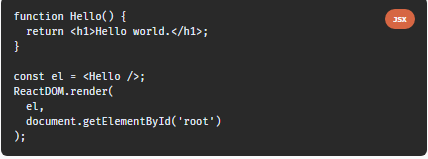
In order to display the component, we need to create the corresponding JSX element.

For example, for our user-defined component **Hello**:





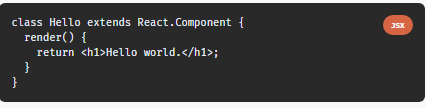
Now, we can use our user-defined element and render it on the page:



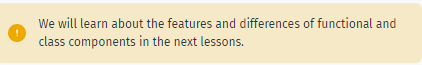


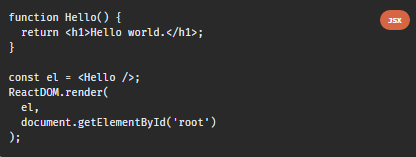
# Class Components

**Class components** are typically used when there are more advanced user interactions, like forms, and animations.  
All class components need to extend the **React. Component** class. We can rewrite our **Hello** functional component as a class component:



Class components need to have a **render**method, which is in charge of telling what the page should show.







## Re-usability

A component used in one area of the application can be reused in another area. This helps speed up the development process

## Nested components

A component can contain several other components

## Difference between class and functional components

|  |  |
| --- | --- |
| Class component | Functional component |
| Regular class extend the component class  They must contain a render method that returns html.  can be contained in a .js or .jsx file | They are javascript functions they return html.  can be contained in a .js or .jsx file |

## Nesting of components

Using import and export methods

## Higher order components

A higher-order component is a function that takes a component and returns a new component it facilitates reusing of component logic

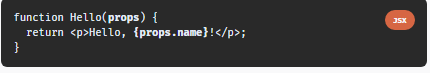
Const NewComponent = higherOrderComponent(orginalComponent);

Pure Components

**Props**

Functional components can accept arguments, similar to JavaScript functions. These arguments are called **props**, and represent an object.

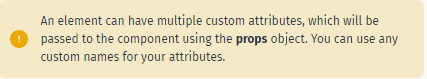
For example, we can use props in our Hello component:

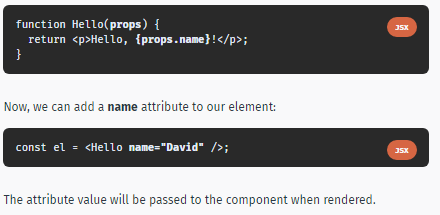


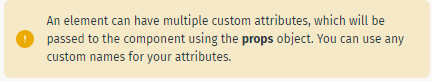
Now, we can add a **name** attribute to our element:



The attribute value will be passed to the component when rendered.

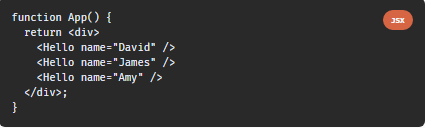




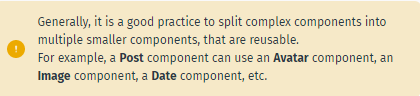


# Components using Components

Components can use other components to generate an output.  
**For example:**

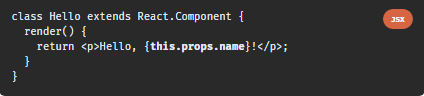


Here, our **App** component uses the **Hello** component three times, each times with a new **name** attribute.

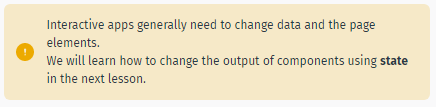


# Props in Class Components

Props can be accessed in class components using **this.props**.  
**For example:**



An important thing to consider is that props are read-only, meaning components cannot modify their props.

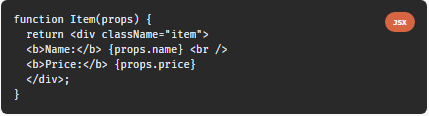


# An Example

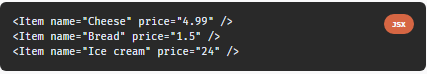
Now that we know how to create components and pass them data, let's create a shopping list. Each item in our list will have a **name** and a **price.**  
**For example:**

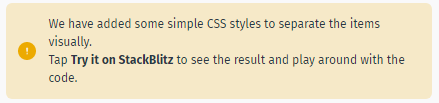


The **Item** component will render a simple div element with the data:



Now we can use our component and create multiple items for our shopping list:

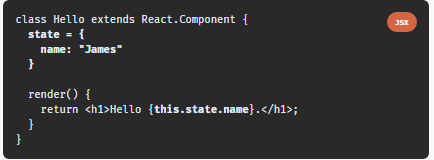




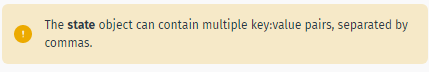
# State

Up until this point, we have learned how to pass data to components using **props**. Many web apps need their components to change their data, for example, after user interaction (clicking a button, submitting a form, etc.). However, **props** cannot be changed.  
In order to allow components to manage and change their data, React provides a feature called **state**.  
**State** is an object that is added as a property in class components.

**For example:**



**[Try it on StackBlitz](https://stackblitz.com/edit/react-state-initial-example?file" \t "_blank)**  
  
as you can see, **state** is just a simple object, that contains key: value pairs.  
Similar to props, the values can be accessed using **this.state**. Now, when the component renders, the state is initialized with the given value and there will be a heading that says "Hello James."



# Changing State

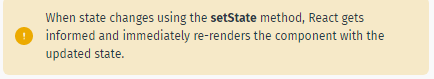
State should not be modified directly. Instead, React provides a **setState()** method, that can be used to modify state.



You need to pass an object with the new key:value pairs to the **setState** method.

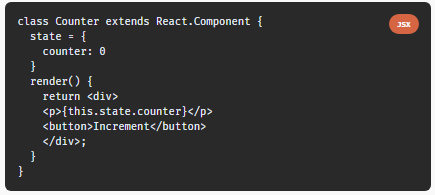
Why should we use **setState**, instead of simply changing the values of the object properties directly?  
The answer uncovers one of the most useful features of React: when **setState** is called, React automatically re-renders the affected component with the new state!

Usually, the change in state happens in event handlers. We will look at an example in the next part!

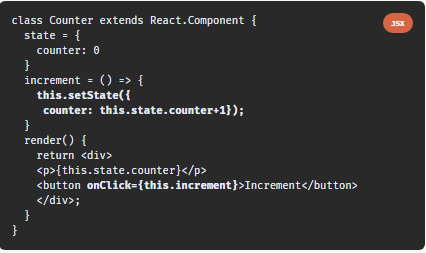


# Counter App

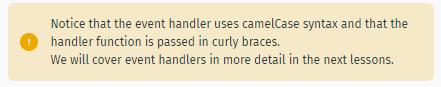
To better understand how state works, let's create a counter app, which increments the counter each time a button is clicked. We start by creating our **Counter** component, which includes the counter and a button:



We have initialized our **counter** to the value 0 in the state.  
Now, we need to add a click event handler to the button and increment the counter in the state.  
Here is the final code:

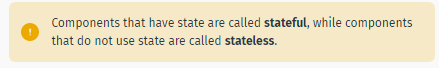


The onClick event calls the **increment** function of our component, which uses **setState** to change the value of our counter. When the state is changed, React automatically triggers a re-render of the component. Tap **Try it on StackBlitz** to see the counter in action!



# Props vs State

As a recap, here is a summary of the main differences between **props** and **state**:  
  
- We use **props** to pass data to components.  
- Components use **state** to manage their data.  
- Props are read-only and cannot be modified.  
- State can be modified by its component using the **setState()** method.  
- The **setState()** method results in re-rendering the component affected.



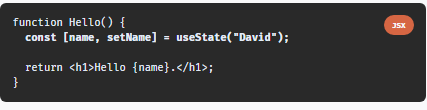
**Hooks**

Earlier version of React allowed to use state only with class components.  
In recent iterations of React, a new feature called **hooks**was introduced, allowing to use state inside of functional components.

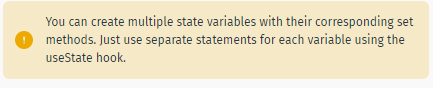
First, we need to import the **useState** hook:



**useState** returns a pair, the current state value and a function, that lets you change the state.  
**useState** takes one argument, which is the initial value of the state.  
  
Let's look at an example:



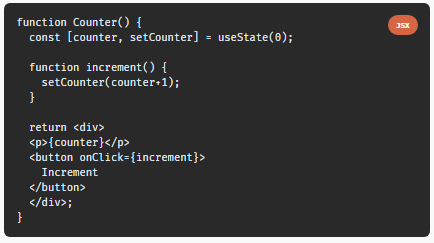
In the example above, we create a **name** state variable and a **setName** function. The square brackets syntax is called **array destructuring**. It assigns the **name** variable to the current state value, and **setName** to the function that allows to change the state. You can name these variables anything you like.  
Then, we pass "David" as the initial value for our name variable to **useState()**.



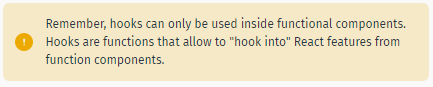
# Counter App using Hooks

Now we can rewrite our Counter app from the previous lesson using a functional component and hooks!

**Here is the code:**



As you can see, compared to the class component, the code is much shorter and easier to read and understand. That was one of the reasons why the React team created Hooks.



# Lifecycle Methods

React provides special lifecycle methods for class components, which are called when components are mounted, updated or unmounted. **Mounting** is the process when a component is rendered on the page. **Unmounting** is the process when a component is removed from the page. The **componentDidMount** method is called when a component is rendered on the page.  
For example, we can use **componentDidMount** in our Counter app to set the initial value of the counter:



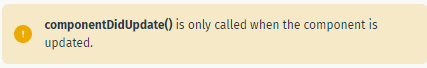
This will set an initial value of the counter when the component is rendered. **componentDidMount** is typically used for populating the state inside of a component when it initially mounts to the DOM.



# componentDidUpdate

Another lifecycle method is **componentDidUpdate()**, which is called when a component is updated in the DOM.  
We can, for example, alert the current counter value when it is incremented:

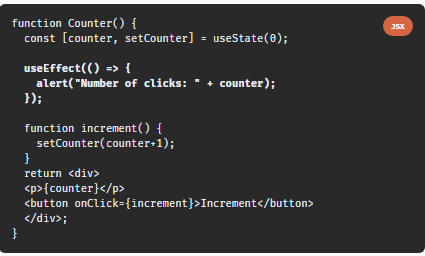




# The useEffect Hook

The lifecycle methods we covered are only available for class components.  
However, React provides a special Hook called **useEffect** to make lifecycle methods available in functional components. It combines the **componentDidMount**, **componentDidUpdate**, and **componentWillUnmount** methods into one.

For example, we can achieve the behavior of our last example using a functional Counter component:



When you run the code, you'll notice that the alert dialog appears also during the first render.

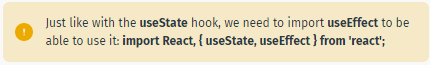
This is caused by the fact that, by default, **useEffect** runs both, after the first render and after every update.  
To call the method only when something changes, we can provide it a second argument:



Now, the useEffect() method will run only if **count** changes.  
To mimic **componentWillUnmount**, useEffect may return a function that cleans up after it:

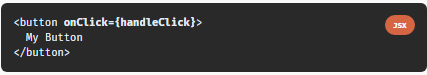


You can have multiple effects in the same component.

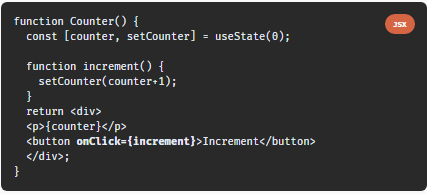


# Event Handling

Handling events in React is very similar to handling events in the DOM.  
The only difference is that event names use camelCase syntax and the event handler needs to be passed in curly braces.  
**For example, to handle the click event on a button:**



Clicking the button will call the **handleClick** function of the component.  
**Let's explore our Counter app:**

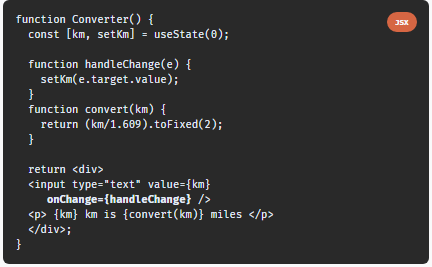


The **onClick** event calls the **increment** function, which is incrementing the counter state variable.

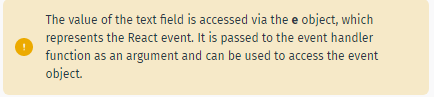


# Handling User Input

One of the common ways that users interact with web pages is through text fields.  
We can handle user input in React using the **onChange** event of the text field.  
When the value of the text field changes, the event handler is called, updating the value of the field in the component's state. This way you always have the actual value of the text field in the state.  
Let's make an app to convert Km to Miles. We will take the Km value from a text field and calculate the miles value upon input:

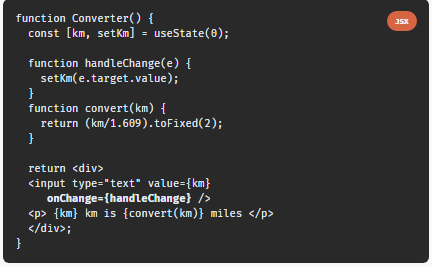


Our Converter component includes a text field, which calls the **handleChange** function when its value changes.  
The handleChange function updates the state with the current value of the textfield, causing the component to re-render and show the corresponding miles value, which is calculated using the **convert** function..

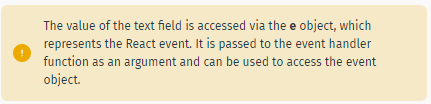


# Handling User Input

One of the common ways that users interact with web pages is through text fields.  
We can handle user input in React using the **onChange** event of the text field.  
When the value of the text field changes, the event handler is called, updating the value of the field in the component's state. This way you always have the actual value of the text field in the state.  
Let's make an app to convert Km to Miles. We will take the Km value from a text field and calculate the miles value upon input:

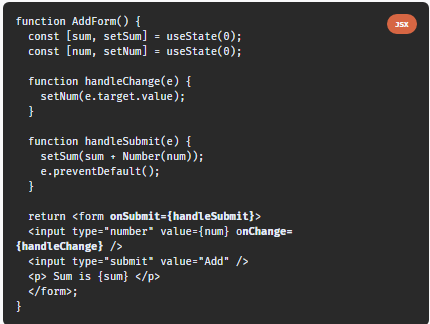


Our Converter component includes a text field, which calls the **handleChange** function when its value changes.  
The handleChange function updates the state with the current value of the textfield, causing the component to re-render and show the corresponding miles value, which is calculated using the **convert** function.

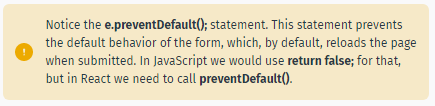


# Forms

In the previous part, we learned how to handle user input in text fields. Text fields are usually part of a form.  
Similar to the previous example, React form elements keep their state and update it based on user input.  
This way you always have the data of your form at your disposal in the state.  
To demonstrate this, we will create a form that will add numbers every time the form is submitted and display the sum.  
Our form contains an input field and a submit button:



In the code above, the value of the input is controlled by React (we keep the value in the state).  
When the form is submitted using the submit button, the **handleSubmit** function gets called, which updates the value of **sum** in the state.  
An input form element whose value is controlled by React in this way is called a "**controlled component**".



# Lists

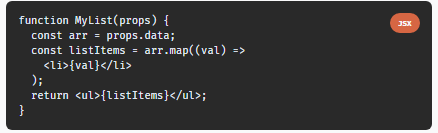
Web apps commonly contain repeating elements, such as lists or sections, where the same DOM element is repeated with a different data set. Consider an array of strings:



We need to render a list <li> element for each item in the array.  
We can define a **MyList** component and pass it the array as a prop using a custom **data** attribute:

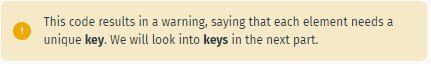


Now, when the array is accessible via props, we can write the component logi c:



We take the input array from the incoming props, loop through the array using the JavaScript **map** function and return a <li> element for each item.  
The resulted array is stored in the **listItems** variable.  
Then, the component returns the listItems array inside a <ul> tag.  
 **The result:**





# Keys

Each element in a list must have a **key** attribute.  
Keys act as a unique identity, identifying each element.  
Usually, these are IDs from your data, or can be auto-generated indexes.

**For example:**

