

# The Life of a Component

There are two ways to define a custom component, both accomplishing the same result but using different syntax:

- **Using a function (components created this way are referred to as function components)**
- **Using a class that extends `React.Component` (commonly referred to as class components)**

# A Custom Function Component

```
const MyComponent = function() {  
  return 'I am so custom';  
};
```

The custom component is just a function that returns the UI that you want. In this case, the UI is only text but you'll often need a little bit more.

```
const MyComponent = function() {  
  return React.createElement('span', null, 'I am so custom');  
};
```

```
ReactDOM.render(  
  MyComponent(),  
  document.getElementById('app')  
);
```

## A JSX Version

The same example using JSX will look a little easier to read. Defining the component looks like this:

```
const MyComponent = function() {  
  return <span>I am so custom</span>;  
};
```

```
ReactDOM.render(  
  <MyComponent />,  
  document.getElementById('app')  
);
```

# A JSX Version

Arrow function syntax

```
function MyComponent() {  
  return (  
    <span>I am so Custom</span>  
  );  
}
```

# A Custom Class Component

The second way to create a component is to define a class that extends `React.Component` and implements a `render()` function:

```
class MyComponent extends React.Component {  
  render() {  
    return React.createElement('span', null, 'I am so custom');  
    // or with JSX:  
    // return <span>I am so custom</span>;  
  }  
}
```

# A Custom Class Component

Rendering the component on the page:

```
ReactDOM.render(  
  React.createElement(MyComponent),  
  document.getElementById('app')  
);
```

If you use JSX, you don't need to know how the component was defined (using a class or a function). In both cases using the component is the same:

```
ReactDOM.render(  
  <MyComponent />,  
  document.getElementById('app')  
);
```

# Properties

- Rendering hard-coded UI in your custom components is perfectly fine and has its uses.
- But the components can also take properties and render or behave differently, depending on the values of the properties.
- Think about the element in HTML and how it acts differently based on the value of the href attribute.
- The idea of properties in React is similar

```
class MyComponent extends React.Component {  
  render() {  
    return <span>My name is <em>{this.props.name}</em></span>;  
  }  
}
```

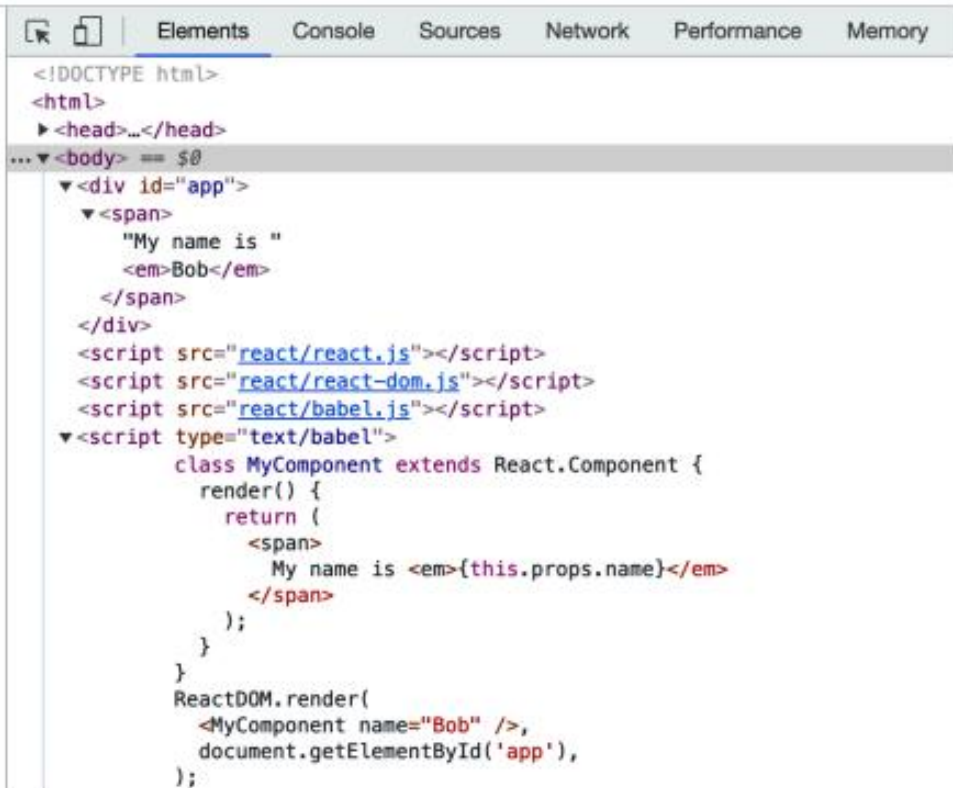
# Properties

Passing a value for the name property when rendering the component looks like this:

```
ReactDOM.render(  
  <MyComponent name="Bob" />,  
  document.getElementById('app')  
);
```



My name is *Bob*



The screenshot shows a web browser's developer tools interface. The top bar includes tabs for Elements, Console, Sources, Network, Performance, and Memory. The 'Elements' tab is active, displaying a tree view of the document's DOM. The root element is `<html>`, which contains a `<head>` and a `<body>`. The `<body>` element is expanded, showing a `<div id="app">` element. Inside this `<div>`, there is a `<span>` element containing the text "My name is " followed by an `<em>Bob</em>` element. Below the `<div>`, there are three `<script>` tags for `react/react.js`, `react/react-dom.js`, and `react/babel.js`. The last `<script>` tag is expanded, showing a JavaScript class `MyComponent` that extends `React.Component`. The `render()` method of `MyComponent` returns a JSX element: `<span>My name is <em>{this.props.name}</em></span>`. The `ReactDOM.render()` function is called with `MyComponent` and the `name="Bob"` prop, targeting the `app` element in the document.

```
<!DOCTYPE html>
<html>
  <head>...</head>
  <body> == $0
    <div id="app">
      <span>
        "My name is "
        <em>Bob</em>
      </span>
    </div>
    <script src="react/react.js"></script>
    <script src="react/react-dom.js"></script>
    <script src="react/babel.js"></script>
    <script type="text/babel">
      class MyComponent extends React.Component {
        render() {
          return (
            <span>
              My name is <em>{this.props.name}</em>
            </span>
          );
        }
      }
      ReactDOM.render(
        <MyComponent name="Bob" />,
        document.getElementById('app'),
      );
    </script>
  </body>
</html>
```

# Properties in Function Components

In function components, there's no `this`, or `this` refers to the global object. So instead of `this.props`, you get a `props` object passed to your function as the first argument:

```
const MyComponent = function(props) {  
  return <span>My name is <em>{props.name}</em></span>;  
};
```

# Properties in Function Components

A common pattern is to use JavaScript's destructuring assignment and assign the property values to local variables. In other words the preceding example becomes:

```
const MyComponent = function({name}) {  
  return <span>My name is <em>{name}</em></span>;  
};
```

# Properties in Function Components

You can have as many properties as you want. If, for example, you need two properties (name and job), you can use them like:

```
const MyComponent = function({name, job}) {  
  return <span>My name is <em>{name}</em>, the {job}</span>;  
};  
ReactDOM.render(  
  <MyComponent name="Bob" job="engineer"/>,  
  document.getElementById('app')  
);
```

# Default Properties

Your component may offer a number of properties, but sometimes a few of the properties may have default values that work well for the most common cases. You can specify default property values using `defaultProps` property for both function and class components.

```
const MyComponent = function({name, job}) {  
  return <span>My name is <em>{name}</em>, the {job}</span>;  
};  
MyComponent.defaultProps = {  
  job: 'engineer',  
};  
ReactDOM.render(  
  <MyComponent name="Bob" />,  
  document.getElementById('app')  
);
```

# Default Properties

Class component:

```
class MyComponent extends React.Component {  
  render() {  
    return (  
      <span>My name is <em>{this.props.name}</em>,  
      the {this.props.job}</span>  
    );  
  }  
}  
MyComponent.defaultProps = {  
  job: 'engineer',  
};  
ReactDOM.render(  
  <MyComponent name="Bob" />,  
  document.getElementById('app')  
);
```

# State

The examples so far were pretty static (or “stateless”)

But where React really shines is when the data in your application changes.

React has the concept of state, which is any data that components want to use to render themselves. When state changes, React rebuilds the UI in the DOM without you having to do anything.

Similarly to how you access properties via **this.props**, you read the state via the object **this.state**. To update the state, you use **this.setState()**.

When **this.setState()** is called, React calls the render method of your component (and all of its children) and updates the UI..

```
class TextAreaCounter extends React.Component {  
  render() {  
    const text = this.props.text;  
    return (  
      <div>  
        <textarea defaultValue={text}/>  
        <h3>{text.length}</h3>  
      </div>  
    );  
  }  
}  
TextAreaCounter.defaultProps = {  
  text: 'Count me as I type',  
};
```



```
ReactDOM.render(  
  <TextAreaCounter text="Bob" />,  
  document.getElementById( 'app' )  
);
```

# Make It Stateful

let's have the component maintain some data (state) and use this data to render itself initially and later on update itself (re-render) when data changes.

First, you need to set the initial state in the class constructor using `this.state`

If you don't do it, consecutive access to `this.state` in the `render()` method will fail

# Make It Stateful

```
class TextAreaCounter extends React.Component {  
  constructor() {  
    super();  
    this.state = {};  
  }  
  render() {  
    const text = 'text' in this.state ? this.state.text : this.props.text;  
    return (  
      <div>  
        <textarea defaultValue={text} />  
        <h3>{text.length}</h3>  
      </div>  
    );  
  }  
}
```

# Make It Stateful

You always update the state with `this.setState()`.

```
onTextChange(event) {  
  this.setState({  
    text: event.target.value,  
  });  
}
```

# Make It Stateful

The last thing left to do is update the render() method to set up the event handler:

```
render() {  
  const text = 'text' in this.state ? this.state.text : this.props.text;  
  return (  
    <div>  
      <textarea  
        value={text}  
        onChange={event => this.onTextChange(event)}  
      />  
      <h3>{text.length}</h3>  
    </div>  
  );  
}
```

```
class TextAreaCounter extends React.Component {
  constructor() {
    super();
    this.state = {};
  }

  render() {
    const text = 'text' in this.state ? this.state.text : this.props.text;
    return (
      <div>
        <textarea
          value={text}
          onChange={(event) => this.onTextChange(event)} // Add this line
        />
        <h3>{text.length}</h3>
      </div>
    );
  }

  onTextChange(event) {
    this.setState({
      text: event.target.value,
    });
  }
}
```

## A Note on DOM Events

```
onChange={event => this.onChange(event)}
```

```
<button onClick="doStuff">
```

the event listener is right there with the UI code)

it's inefficient to have too many event listeners scattered like this

```
<div id="parent">
  <button id="ok">OK</button>
  <button id="cancel">Cancel</button>
</div>
```

```
<script>
```

```
document.getElementById('parent').addEventListener('click', function(event) {
  const button = event.target;
```

```
// do different things based on which button was clicked
```

```
switch (button.id) {
```

```
  case 'ok':
```

```
    console.log('OK!');
```

```
    break;
```

```
  case 'cancel':
```

```
    console.log('Cancel');
```

```
    break;
```

```
  default:
```

```
    new Error('Unexpected button ID');
```

```
};
```

```
});
```

```
</script>
```



- This works and performs fine, but there are drawbacks
- Declaring the listener is further away from the UI component, which makes code harder to find and debug.
- Using delegation and always switch-ing creates unnecessary boilerplate code even before you get to do the actual work (responding to a button click in this case).
- Browser inconsistencies (omitted here) actually require this code to be longer.

Unfortunately, when it comes to taking this code live in front of real users, you need a

few more additions if you want to support old browsers:

- You need `attachEvent` in addition to `addEventListener`.
- You need `const event = event || window.event;` at the top of the listener.
- You need `const button = event.target || event.srcElement;`

# Props Versus State

you have access to `this.props` and `this.state` when it comes to displaying your component in your `render()` method. You may be wondering when you should use one versus the other.

**Properties are a mechanism for the outside world (users of the component) to configure your component.**

**State is your internal data maintenance.**

consider an analogy with object-oriented programming, **`this.props` is like a collection of all the arguments passed to a class constructor, while `this.state` is a bag of your private properties.**

# Props in Initial State: an Antipattern

In the preceding textarea example, it's tempting to use `this.props` to set the initial `this.state`:

```
// Warning: Anti-pattern  
this.state = {  
  text: props.text,  
};
```

This is considered an antipattern. Ideally, you use any combination of `this.state` and `this.props` as you see fit to build your UI.

But sometimes you want to take a value passed to your component and use it to construct the initial state.

There's nothing wrong with this, except that the callers of your component may expect the property (text here) to always have the latest value, and the preceding code would violate this expectation.

# Props in Initial State: an Antipattern

To set the expectation straight, a simple naming change is sufficient—for example, calling the property something like `defaultText` or `initialValue` instead of just `text`:

```
this.state = {  
  text: props.defaultText, // Use a different property name  
};
```

The component now clearly communicates that it uses an initial value (not necessarily the latest prop value).

# Accessing the Component from the Outside

You don't always have the luxury of starting a brand new React app from scratch.

Sometimes you need to hook into an existing application or a website and migrate to React one piece at a time.

One way your React app can communicate with the outside world is to get a reference to a component you render with ReactDOM.render() and use it from outside of the component.

```
const myTextAreaCounter = ReactDOM.render(  
  <TextAreaCounter text="Bob" />,  
  document.getElementById('app')  
);
```

# Accessing the Component from the Outside

Now you can use `myTextAreaCounter` to access the same methods and properties you normally access with this when inside the component

```
> myTextAreaCounter.state
```

```
< ▶ {}
```

```
> myTextAreaCounter.props
```

```
< ▶ {text: 'Bob'}
```

```
> myTextAreaCounter.setState({text: 'Bob, Sponge'})
```

```
myTextAreaCounter.setState({text: "Hello outside world!"});
```

This line gets a reference to the main parent DOM node that React created:

```
const reactAppNode = ReactDOM.findDOMNode(myTextAreaCounter);
```

This is the first child of the `<div id="app">`, which is where you told React to do its magic.



# Lifecycle Methods

React offers several so-called lifecycle methods. You can use the lifecycle methods to

listen to changes in your component as far as the DOM manipulation is concerned.

The life of a component goes through three steps:

- **Mounting**

The component is added to the DOM initially.

- **Updating**

The component is updated as a result of calling `setState()` and/or a prop provided to the component has changed.

- **Unmounting**

The component is removed from the DOM.

## Mounting Phase:

**constructor():** Called before anything else. Set up initial state and other values.

**getDerivedStateFromProps():** Invoked right before rendering. Use it to update state based on initial props.

**render():** Required method that outputs HTML to the DOM.

**componentDidMount():** Called after rendering. Use it for actions that require the component to be in the DOM.

## Updating Phase:

**shouldComponentUpdate():** Decides whether the component should re-render. Optimize performance.

**render():** Re-renders the component.

**getSnapshotBeforeUpdate():** Captures information before changes are applied to the DOM.

**componentDidUpdate():** Called after updates are applied. Perform side effects here.

# Unmounting Phase

**componentWillUnmount():** Cleanup tasks (e.g., removing event listeners) before unmounting.

After the initial mounting and after the commit to the DOM, your component's **componentDidMount()** method is called.

It's called after a component has been added to the DOM (i.e., mounted). Here's what it does:

Purpose:

Fetching data from APIs.

Setting up subscriptions or event listeners.

Modifying the DOM (e.g., initializing third-party libraries).

```
class MyComponent extends React.Component {  
  componentDidMount() {  
    // Fetch data from an API  
    fetch('https://api.example.com/data')  
      .then(response => response.json())  
      .then(data => {  
        // Update component state with fetched data  
        this.setState({ data });  
      });  
  }  
}
```

**componentWillUnmount()** is called. This is the place to do any cleanup work you may need. Any event handlers, or anything else that may leak memory, should be cleaned up here.

```
componentWillUnmount() {  
  // Clean up event listener  
  window.removeEventListener('resize', this.handleResize);  
}
```

Before the component is updated (e.g., as a result of `setState()`), you can use **`getSnapshotBeforeUpdate()`**. This method receives the previous properties and state as arguments.

**`componentDidUpdate()`**. You can use this information to compare the old and the new state and potentially make more network requests if necessary.



```
class MyComponent extends React.Component {  
  getSnapshotBeforeUpdate(prevProps, prevState) {  
    // Capture some information (e.g., scroll position)  
    return document.documentElement.scrollTop;  
  }  
  
  componentDidUpdate(prevProps, prevState, snapshot) {  
    // Use the captured information (e.g., restore scroll position)  
    if (snapshot !== null) {  
      document.documentElement.scrollTop = snapshot;  
    }  
  }  
  
  render() {  
    // Render component content  
  }  
}
```

And then there's **shouldComponentUpdate(newProps, newState)**, which is an opportunity for an optimization. You're given the state-to-be, which you can compare with the current state and decide not to update the component, in which case its `render()` method is not called.

```
class MyComponent extends React.Component {  
  shouldComponentUpdate(newProps, newState) {  
    // Compare new props or state with current ones  
    if (newProps.someProp === this.props.someProp && newState.someState === this.state.someState)  
    {  
      return false; // No need to update  
    }  
    return true; // Proceed with update  
  }  
  
  render() {  
    // Render component content  
  }  
}
```

```
class TextAreaCounter extends React.Component {  
  // ...  
  
  componentDidMount() {  
    console.log('componentDidMount');  
  }  
  componentWillUnmount() {  
    console.log('componentWillUnmount');  
  }  
  componentDidUpdate(prevProps, prevState, snapshot) {  
    console.log('componentDidUpdate', prevProps, prevState, snapshot);  
  }  
  getSnapshotBeforeUpdate(prevProps, prevState) {  
    console.log('getSnapshotBeforeUpdate', prevProps, prevState);  
    return 'hello';  
  }  
  shouldComponentUpdate(nextProps, nextState) {  
    console.log('shouldComponentUpdate', nextProps, nextState);  
    return true;  
  }  
  
  // ...  
}
```

After loading the page, the only message in the console is **componentDidMount** when you type “b” to make the text “Bobb” **shouldComponentUpdate()** is called with the new props.

Since this method returns true, React proceeds with calling **getSnapshotBeforeUpdate()** passing the old props and state.

Finally, **componentDidUpdate()** is called with the old info and any snapshot defined by the previous method.

# Paranoid State Protection

Say you want to restrict the number of characters to be typed in the textarea. You should do this in the event handler `onTextChange()`, which is called as the user types. But what if someone calls `setState()` from the outside of the component? Can you still protect the consistency and well-being of your component? Sure. You can do the validation in `componentDidUpdate()` and if the number of characters is greater than allowed,

## Paranoid State Protection

```
componentDidUpdate(prevProps, prevState) {  
  if (this.state.text.length > 3) {  
    this.setState({  
      text: prevState.text || this.props.text,  
    });  
  }  
}
```

# Lifecycle Example: Using a Child Component

you can mix and nest React components as you see fit

This is the parent:

```
class LifecycleLoggerComponent extends React.Component {  
  static getName() {}  
  componentDidMount() {  
    console.log(this.constructor.getName() + '::componentDidMount');  
  }  
  componentWillUnmount() {  
    console.log(this.constructor.getName() + '::componentWillUnmount');  
  }  
  componentDidUpdate(prevProps, prevState, snapshot) {  
    console.log(this.constructor.getName() + '::componentDidUpdate');  
  }  
}
```

## Lifecycle Example: Using a Child Component

```
class Counter extends LifecycleLoggerComponent {  
  static getName() {  
    return 'Counter';  
  }  
  render() {  
    return <h3>{this.props.count}</h3>;  
  }  
}  
Counter.defaultProps = {  
  count: 0,  
};
```



## Lifecycle Example: Using a Child Component

```
class TextAreaCounter extends LifecycleLoggerComponent {  
    static getName() {  
        return 'TextAreaCounter';  
    }  
    // ....  
}
```

# Lifecycle Example: Using a Child Component

```
render() {  
  const text = 'text' in this.state ? this.state.text : this.props.text;  
  return (  
    <div>  
      <textarea  
        value={text}  
        onChange={this.onTextChange}  
      />  
      {text.length > 0  
        ? <Counter count={text.length} />  
        : null  
      }  
    </div>  
  );  
}
```

Notice the conditional statement in JSX. You wrap the expression in `{}` and conditionally render either or nothing (null).

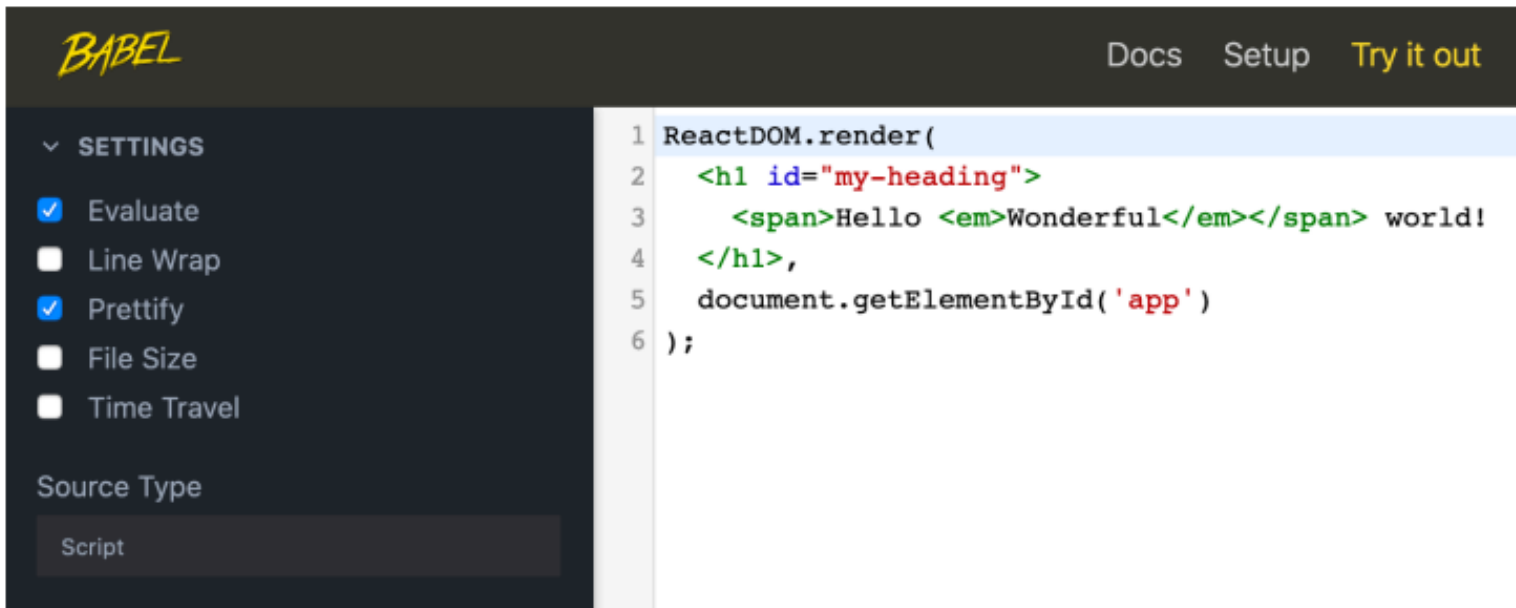
And just for demonstration: another option is to move the condition outside the return. Assigning the result of a JSX expression to a variable is perfectly fine.

```
render() {  
  const text = 'text' in this.state  
    ? this.state.text  
    : this.props.text;  
  let counter = null;  
  if (text.length > 0) {  
    counter = <Counter count={text.length} />;  
  }  
  return (  
    <div>  
      <textarea  
        value={text}  
        onChange={this.onTextChange}  
      />  
      {counter}  
    </div>  
  );  
}
```

# JSX

# A Couple Tools

To experiment and get familiar with the JSX transforms, you can play with the live editor at <https://babeljs.io/repl> . Make sure you check the “Prettify” option for better readability of the result.



The screenshot shows the Babel REPL interface. On the left, the 'SETTINGS' panel is open, showing the 'Prettify' option checked. The 'Source Type' is set to 'Script'. The main editor area displays the following code:

```
1 ReactDOM.render(  
2   <h1 id="my-heading">  
3     <span>Hello <em>Wonderful</em></span> world!  
4   </h1>,  
5   document.getElementById( 'app' )  
6 );
```

the JSX source of “Hello world!” from Chapter 1 becomes a series of calls to `React.createElement()`, using the function syntax React works with.

[Docs](#)[Setup](#)[Try it out](#)[Videos](#)[Blog](#)[Search](#)[Donate](#)[Team](#)[GitHub](#)

```
1 ReactDOM.render(  
2   <h1 id="my-heading">  
3     <span>Hello <em>Wonderful</em></span> world!  
4   </h1>,  
5   document.getElementById('app')  
6 );
```

```
1 "use strict";  
2  
3 ReactDOM.render(  
4   /*#__PURE__*/ React.createElement(  
5     "h1",  
6     {  
7       id: "my-heading"  
8     },  
9     /*#__PURE__*/ React.createElement(  
10      "span",  
11      null,  
12      "Hello ",  
13      /*#__PURE__*/ React.createElement("em", null, "Wonderful")  
14    ),  
15    " world!"  
16  ),  
17  document.getElementById("app")  
18 );  
19
```

Another online tool you may find helpful when learning JSX or transitioning an existing app's markup from HTML is the HTML-to-JSX compiler

## HTML to JSX Compiler

☐ Create class

Live HTML Editor

```
<!-- Hello world -->
<div class="awesome" style="border: 1px solid red">
  <label for="name">Enter your name: </label>
  <input type="text" id="name" />
</div>
<p>Enter your HTML here</p>
```

```
<div>
  { /* Hello world */ }
  <div className="awesome" style={{border: '1px solid red'}}>
    <label htmlFor="name">Enter your name: </label>
    <input type="text" id="name" />
  </div>
  <p>Enter your HTML here</p>
</div>
```



# Whitespace in JSX

Whitespace in JSX is similar to HTML but not identical. For example, if you have this JSX:

```
function Example1() {  
  return (  
    <h1>  
      {1} plus {2} is {3}  
    </h1>  
  );  
}
```

# Whitespace in JSX

When React renders it in the browser (you can inspect the resulting HTML in the browser's dev tools), the generated HTML looks like this:

```
<h1>1 plus 2 is 3</h1>
```

This is effectively an h1 DOM node with five children

which renders as **“1 plus 2 is 3.”**

Exactly as you'd expect in HTML, multiple spaces become one when rendered in the browser

However, in this next example

```
function Example2() {  
  return (  
    <h1>  
      {1}  
      plus  
      {2}  
      is  
      {3}  
    </h1>  
  );  
}
```

...you end up with:

```
<h1>  
  1plus2is3  
</h1>
```

```
function Example3() {  
  return (  
    <h1>  
      { /* space expressions */  
        {1}  
        {' '}plus{' '}  
        {2}  
        {' '}is{' '}  
        {3}  
      </h1>  
    );  
}
```

```
function Example4() {  
  return (  
    <h1>  
      { /* space glued to string expressions */  
        {1}  
        {' plus '  
        {2}  
        {' is '  
        {3}  
      </h1>  
    );  
}
```

# Comments in JSX

Because the expressions wrapped in `{}` are just JavaScript, you can easily add multi- line comments using `/* comment */`. You can also add single-line comments using `// comment`.

```
<h1>
  {/* multiline comment */}
  {/*
    multi
    line
    comment
    */}
  {
    // single line
  }
  Hello!
</h1>
```

# HTML Entities

You can use HTML entities in JSX like so:

```
<h2>
```

```
  More info &raquo;
```

```
</h2>
```

This example produces a “double right angle quote,”

**More info »**

# HTML Entities

However, if you use the entity as part of an expression, you will run into double encoding issues

```
<h2>  
  {"More info &raquo;"}  
</h2>
```

**More info &raquo;**



# HTML Entities

To prevent the double-encoding, you can use the Unicode version of the HTML entity, which in this case is `\u00bb`

(<https://dev.w3.org/html5/html-author/charref>):

```
<h2>  
  {"More info \u00bb"}  
</h2>
```

For convenience, you can define a constant somewhere at the top of your module together with any common spacing. For example:

```
const RAQUO = ' \u00bb';
```

# HTML Entities

Then use the constant anywhere you need, for example:

```
<h2>  
  {"More info" + RAQUO}  
</h2>  
<h2>  
  {"More info"}{RAQUO}  
</h2>
```

# Spread Attributes

JSX borrows a feature from ECMAScript called the spread operator and adopts it as a convenience when defining properties.

Imagine you have a collection of attributes you want to pass to an component

```
const attr = {  
  href: 'https://example.org',  
  target: '_blank',  
};
```

You can always do it like so:

```
return (  
  <a  
    href={attr.href}  
    target={attr.target}>  
    Hello  
  </a>  
);
```

# Spread Attributes

By using spread attributes, you can accomplish this in just one line:

```
return <a {...attr}>Hello</a>;
```

# Returning Multiple Nodes in JSX

You always have to return a single node (or an array) from your render function. Returning two nodes is not allowed. In other words, this is an error:

# Returning Multiple Nodes in JSX

```
// Syntax error:  
// Adjacent JSX elements must be wrapped in an enclosing tag  
function InvalidExample() {  
  return (  
    <span>  
      Hello  
    </span>  
    <span>  
      World  
    </span>  
  );  
}
```

# A Wrapper

The fix is to just wrap all the nodes in another component such as a `<div>` (and add a space between the “Hello” and “World” while you’re at it):

```
<div>
  <span>Hello</span>
  { ' ' }
  <span>World</span>
</div>
```



# A Fragment

To remove the need for an extra wrapper element, newer versions of React added fragments, which are wrappers that do not add additional DOM nodes when the component is rendered.

```
function FragmentExample() {  
  return (  
    <React.Fragment>  
      <span>Hello</span>  
      {' '  
      <span>World</span>  
    </React.Fragment>  
  );  
}
```

# A Fragment

Furthermore, the `React.Fragment` part can be omitted and these empty elements also work:

```
function FragmentExample() {  
  return (  
    <>  
      <span>Hello</span>  
      { ' ' }  
      <span>World</span>  
    </>  
  );  
}
```

# An Array

Another option is to return an array of nodes, as long as the nodes in the array have proper key attributes. Note the required commas after every element of the array.

## An Array

```
function ArrayExample() {  
  return [  
    <span key="a">Hello</span>,  
  
    ' ',  
    <span key="b">World</span>,  
    '!',  
  ];  
}
```