**Babel** is a compiler, which converts jsx code into ES5 JavaScript that all browser understands.

**Webpack** is a tool which that is watching files to change and when they do it feeds those files into Babel, which turns jsx into Js.

**Return without brackets will give error if we use wrapper html element in next line of return**

function HelloWorld() {

return

<div>

<Hello/> <World/>!

</div>;

}

This will fail with an error.

**Correct way**

function HelloWorld() {

return <div>

<Hello/> <World/>!

</div>;

}

**Best way Wrap With a Tag**

**A component function must return a single element.**

function HelloWorld() {

return (

<div>

<Hello/> <World/>!

</div>

);

}

A lot of the time, this is perfectly fine. But sometimes, you won’t want to have a wrapper element, like

if you have a component that returns two table cells:

function NameCells() {

return (

<td>First Name</td>

<td>Last Name</td>

);

}

You can’t wrap these elements in a <div>, because the <td> table cells need to be direct descendants

of a <tr> table row. How can you combine them?

**Fragments**

React’s answer is the *fragment*. This component was added in React 16.2, and can be used like this:

function NameCells() {

return (

<React.Fragment>

<td>First Name</td>

<td>Last Name</td>

</React.Fragment>

);

}

After rendering, the React.Fragment component will “disappear”, leaving only the children inside it,

so that the DOM structure will have no wrapper components.

Fragments make it easier to produce valid HTML (such as keeping <td> elements directly inside

<tr>s), and they keep the DOM structure flatter which makes it easier to write semantic HTML

(which is also usually more accessible HTML).

**Fragment Syntax use empty tag**

function NameCells() {

return (

<>

<td>First Name</td>

<td>Last Name</td>

</>

);

}

**“If” in JSX**

The next question you might wonder is, “How do I write a conditional if I can’t use ‘if’?” There are a

couple of options.

The first is the ternary operator (the question mark, ?). Use it like this:

33

function ValidIndicator() {

const isValid = true;

return (

<span>{isValid ? 'valid' : 'not valid'}</span>

);

}

**You can also use Boolean operators such as && like this:**

function ValidIndicator() {

const isValid = true;

return (

<span>

{isValid && 'valid'}

{!isValid && 'not valid'}

</span>

);

}

**Capitalize Component Names**

The components you write must begin with an uppercase letter. This means using names like

**UserList** and **Menu** and **SubmitButton**, and not names like **userList**, menu, and **submit\_button**.

**In JSX, a component that starts with a lowercase letter is assumed to be a built-in HTML or SVG**

element (div, ul, rect, etc.).

**Concatenation of string and variables by using back quote `**

function Greetings(){

    let username= "root";

    //let username= undefined;

    return(

        <div>

            {username ? `Hello ${username}`:'Not logged in'}

        </div>

    );

**Learn React Testing Tool:**

**1-jest.**

**2-Enzyme.**

**Props**

Where HTML elements have “attributes,” React components have “props” (short for “properties”).

function Dave() {

const firstName = "Dave";

const lastName = "Ceddia";

return (

<Person

className='person'

age={33}

name={firstName + ' ' + lastName} />

);

}

Remember that in JSX, singles braces must surround JavaScript expressions. The code in the braces

is real JavaScript, and it follows all the same scoping rules as normal JavaScript.

It’s important to understand that the JS inside the braces must be an *expression*, not a statement.

Here are a few things you can do inside JSX expressions:

• Math, concatenation: {7 + 5} or {'Your' + 'Name'}

• Function calls: {this.getFullName(person)}

• Ternary (?) operator: {name === 'Dave' ? 'me' : 'not me'}

• Boolean expressions: {isEnabled && 'enabled'}

Here are some things you cannot do:

• Define new variables with let, const, and var

• Use if, for, while, etc.

• Define functions with function

**Communicating with Parent Components**

If you can’t change props, but you need to communicate something up to a parent component, how

does that work?

If a child needs to send data to its parent, the parent can send down a *function* as a prop, like this:

function handleAction(event) {

console.log('Child did:', event);

}

function Parent() {

return (

<Child onAction={handleAction}/>

);

}

The Child component receives the onAction prop, which it can call whenever it needs to send up

data or notify the parent that something happened.

One place where it’s common to pass functions as props is for handling events. For instance, the

built-in button element accepts an onClick prop, which it’ll call when the button is clicked.

function Child({ onAction }) {

return (

<button onClick={onAction}/>

);

}

**PropTypes**

We’ve seen what “props” are, and how they’re passed into React components – but what happens if you forget to pass one of the props?

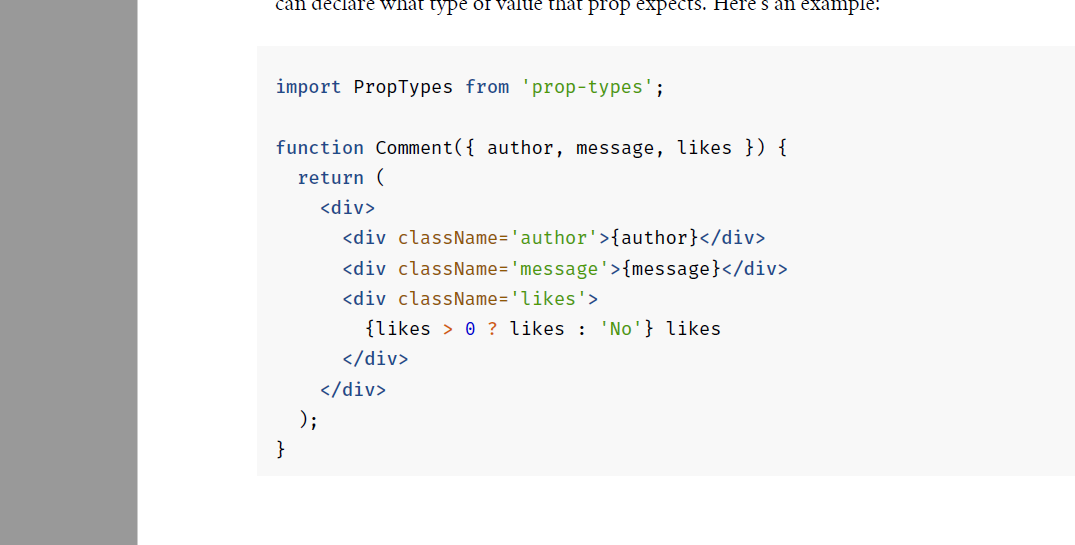
Well, it ends up being undefined, just as if you’d forgotten to pass an argument to a plain old function.

This can be totally fine, or a code-breaking disaster (just as if you’d forgotten to pass an argument to a plain old function).

If you want to avoid this, there are two main options: write your app in TypeScript, or stick with JS and be diligent about using PropTypes.

**How to Write PropTypes**

When you create a component, you can declare that certain props are optional or required, *and* you can declare what type of value that prop expects. Here’s an example:



**State in Classes**

Up until this point we’ve used *props* to pass data to components. But props are *read-only*. What if you

need to keep track of data that can change? What if you want to make your app interactive? This is

where *state* comes in.

**Example: A Counter**

Here is a Parent component that contains a Child component. Parent passes down a function which

Child calls whenever a button is clicked:

function handleAction(event) {

console.log('Child did:', event);

}

function Parent() {

return (

<Child onAction={handleAction}/>

);

}

function Child({ onAction }) {

return (

<button onClick={onAction}>

Click Me!

</button>

);

}

What if we wanted Parent to keep track of how many times the button was clicked? In other words,

Parent should track how many times the handleAction function is called.

To do this, Parent needs to remember the count using *state*, and persist that state between renders.

We’ll want the *initial state* of the counter to be 0, a way to increment the counter, and a way to display

the current count. Plus, whenever the count changes, we’ll need to re-render the app to show the

latest count.

React has two ways of adding state to components. The first is to rewrite the component as a *class*.

Class components have been able to maintain state since the very first versions of React, and every

version of React supports classes.

The second, more modern way, is to use *hooks* to add state directly to a function component

class CountingParent extends React.Component {

// The constructor is called when a

// component is created

constructor(props) {

super(props);

// Set the state here. Use "props" if needed.

this.state = {

actionCount: 0

};

// Bind the event handler function, so that its

// `this` binding isn't lost when it gets passed

// to the button

this.handleAction = this.handleAction.bind(this);

}

handleAction(action) {

console.log('Child says', action);

// Replace actionCount with an incremented value

this.setState({

actionCount: this.state.actionCount + 1

});

}

render() {

return (

<div>

<Child onAction={this.handleAction}/>

<p>Clicked {this.state.actionCount} times</p>

</div>

);

}

}

The Child component doesn’t have to change at all. Try it out! Click the button. Watch it increment.

Ooooh. Aaaah. Such counter.

Note that every instance of a component has its own state. If you have more than one CountingParent

component on the page, each will have its own counter that starts at 0 and increments independently

of the others. You can prove this to yourself by creating a component that contains a few

CountingParents:

const Page = () => (

<div>

<CountingParent/>

<CountingParent/>

<CountingParent/>

</div>

);

**setState Is Asynchronous**

I kinda lied to you up there. I’m sorry. I implied that the setState function would immediately

update the state and call render. That’s not really what happens. The setState function is actually

*asynchronous*.

If you call setState and immediately console.log(this.state) right afterwards, it will very likely

print the old state instead of the one you just set.

// Assume state is { count: 3 }

// Then call setState:

this.setState({ count: 4 });

// Then try to print the "new" state:

console.log(this.state);

// It'll likely print { count: 3 }

// instead of { count: 4 }

If you need to set the state and immediately act on that change, you can pass in a callback function as

the second argument to setState, like this:

this.setState({name: 'Joe'}, function() {

// called after state has been updated

// and the component has been re-rendered

// this.state now contains { name: 'Joe' }

});

**Functional setState**

Another way to make it so that sequential state updates run in sequence is to use the *functional* form

of setState, like this:

this.setState((state, props) => {

return {

value: state.value + 1

}

});

In this form, you pass a *function* to setState instead of an object. The function receives the current

state and props as arguments, and it is expected to return an object, which will be merged with the

old state. If you were to run a few of these sequentially…

this.setState((state, props) => {

return {

value: state.value + 1

}

});

this.setState((state, props) => {

return {

value: state.value + 1

}

});

this.setState((state, props) => {

return {

value: state.value + 1

}

});

This would work as expected, eventually incrementing value by 3.

Functional setState is the preferred way to call setState because it’s guaranteed to work correctly, every

time. Try to use it whenever you can.

**Cleaner Syntax for Class Components**

Earlier I mentioned that classes can be written without a constructor. Let’s see how that works.

Here’s the same example from earlier, with the CountingParent component rewritten without a

constructor:

class CountingParent extends React.Component {

// initialize state with a property initializer

// you can access this.props if needed

state = {

actionCount: 0

};

// writing the handler as an arrow function

// means it will retain the proper value of

// `this`, so we can avoid having to bind it

handleAction = (action) => {

console.log('Child says', action);

// Replace actionCount with an incremented value

this.setState({

actionCount: this.state.actionCount + 1

});

}

render() {

return (

<div>

<Child onAction={this.handleAction}/>

<p>Clicked {this.state.actionCount} times</p>

</div>

);

}

}

**React Naming convention**

Component Name: Start with Capital Letter like Trello, Counter, Parent etc.

React Events Name: *camel* Case like onClick, onSubmit, onKeyDown etc.

Functional Component: Start with Capital Letter like Trello, Counter, Parent etc.

React Events List:

**https://reactjs.org/docs/events.html**

**HTML events are lower cases**

Like onclick, onsubmit etc