**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:

