JSX In Depth

Fundamentally, JSX just provides syntactic sugar for the `React.createElement(component, props, ...children)` function. The JSX code:

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```

compiles into:

```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

You can also use the self-closing form of the tag if there are no children. So:

```
<div className="sidebar" />
```

compiles into:

```
React.createElement(
  'div',
  {className: 'sidebar'},
  null
)
```

If you want to test out how some specific JSX is converted into JavaScript, you can try out the online Babel compiler.

Specifying The React Element Type

The first part of a JSX tag determines the type of the React element.
Capitalized types indicate that the JSX tag is referring to a React component. These tags get compiled into a direct reference to the named variable, so if you use the JSX `<Foo />`expression, `Foo` must be in scope.

React Must Be in Scope

Since JSX compiles into calls to `React.createElement`, the `React` library must also always be in scope from your JSX code.
For example, both of the imports are necessary in this code, even though `React` and `CustomButton` are not directly referenced from JavaScript:

```
import React from 'react';
import CustomButton from './CustomButton';

function WarningButton() {
  // return React.createElement(CustomButton, {color: 'red'}, null);
  return <CustomButton color="red" />;
}
```

If you don't use a JavaScript bundler and loaded React from a `<script>` tag, it is already in scope as the `React` global.

Using Dot Notation for JSX Type

You can also refer to a React component using dot-notation from within JSX. This is convenient if you have a single module that exports many React components. For example, if `MyComponents.DatePicker` is a component, you can use it directly from JSX with:

```
import React from 'react';

const MyComponents = {
  DatePicker: function DatePicker(props) {
    return <div>Imagine a {props.color} datepicker here.</div>;
  }
}

function BlueDatePicker() {
  return <MyComponents.DatePicker color="blue" />;
}
```

User-Defined Components Must Be Capitalized

When an element type starts with a lowercase letter, it refers to a built-in component like `<div>` or `<span>` and results in a string `'div'` or `'span'` passed to `React.createElement`. Types that start with a capital letter like `<Foo />` compile to `React.createElement(Foo)` and correspond to a component defined or imported in your JavaScript file.

We recommend naming components with a capital letter. If you do have a component that starts with a lowercase letter, assign it to a capitalized variable before using it in JSX.

For example, this code will not run as expected:

```
import React from 'react';

// Wrong! This is a component and should have been capitalized:
function hello(props) {
  // Correct! This use of <div> is legitimate because div is a valid HTML tag:
  return <div>Hello {props.toWhat}</div>;
```

```
}

function HelloWorld() {
  // Wrong! React thinks <hello /> is an HTML tag because it's not capitalized:
  return <hello toWhat="World" />;
}
```

To fix this, we will rename `hello` to `Hello` and use `<Hello />` when referring to it:

```
import React from 'react';

// Correct! This is a component and should be capitalized:
function Hello(props) {
  // Correct! This use of <div> is legitimate because div is a valid HTML tag:
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // Correct! React knows <Hello /> is a component because it's capitalized.
  return <Hello toWhat="World" />;
}
```

Choosing the Type at Runtime

You cannot use a general expression as the React element type. If you do want to use a general expression to indicate the type of the element, just assign it to a capitalized variable first. This often comes up when you want to render a different component based on a prop:

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // Wrong! JSX type can't be an expression.
  return <components[props.storyType] story={props.story} />;
}
```

To fix this, we will assign the type to a capitalized variable first:

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};
```

```
function Story(props) {
  // Correct! JSX type can be a capitalized variable.
  const SpecificStory = components[props.storyType];
  return <SpecificStory story={props.story} />;
}
```

Props in JSX

There are several different ways to specify props in JSX.

JavaScript Expressions as Props

You can pass any JavaScript expression as a prop, by surrounding it with `{}`. For example, in this JSX:

```
<MyComponent foo={1 + 2 + 3 + 4} />
```

For `MyComponent`, the value of `props.foo` will be `10` because the expression `1 + 2 + 3 + 4`gets evaluated.

`if` statements and `for` loops are not expressions in JavaScript, so they can't be used in JSX directly. Instead, you can put these in the surrounding code. For example:

```
function NumberDescriber(props) {
  let description;
  if (props.number % 2 == 0) {
    description = <strong>even</strong>;
  } else {
    description = <i>odd</i>;
  }
  return <div>{props.number} is an {description} number</div>;
}
```

You can learn more about conditional rendering and loops in the corresponding sections.

String Literals

You can pass a string literal as a prop. These two JSX expressions are equivalent:

```
<MyComponent message="hello world" />

<MyComponent message={'hello world'} />
```

When you pass a string literal, its value is HTML-unescaped. So these two JSX expressions are equivalent:

```
<MyComponent message="&lt;3" />

<MyComponent message={'<3'} />
```

This behavior is usually not relevant. It's only mentioned here for completeness.

Props Default to “True”

If you pass no value for a prop, it defaults to `true`. These two JSX expressions are equivalent:

```
<MyTextBox autocomplete />

<MyTextBox autocomplete={true} />
```

In general, we don’t recommend using this because it can be confused with the ES6 object shorthand `{foo}` which is short for `{foo: foo}` rather than `{foo: true}`. This behavior is just there so that it matches the behavior of HTML.

Spread Attributes

If you already have `props` as an object, and you want to pass it in JSX, you can use `...` as a “spread” operator to pass the whole props object. These two components are equivalent:

```
function App1() {
  return <Greeting firstName="Ben" lastName="Hector" />;
}

function App2() {
  const props = {firstName: 'Ben', lastName: 'Hector'};
  return <Greeting {...props} />;
}
```

You can also pick specific props that your component will consume while passing all other props using the spread operator.

```
const Button = props => {
  const { kind, ...other } = props;
  const className = kind === "primary" ? "PrimaryButton" : "SecondaryButton";
  return <button className={className} {...other} />;
};

const App = () => {
  return (
    <div>
      <Button kind="primary" onClick={() => console.log("clicked!")}>
        Hello World!
      </Button>
    </div>
  );
};
```

In the example above, the `kind` prop is safely consumed and *is not* passed on to the `<button>` element in the DOM. All other props are passed via the `...other` object making this component really flexible. You can see that it passes an `onClick` and `children` props.

Spread attributes can be useful but they also make it easy to pass unnecessary props to components that don’t care about them or to pass invalid HTML attributes to the DOM. We recommend using this syntax sparingly.

Children in JSX

In JSX expressions that contain both an opening tag and a closing tag, the content between those tags is passed as a special prop: `props.children`. There are several different ways to pass children:

String Literals

You can put a string between the opening and closing tags and `props.children` will just be that string. This is useful for many of the built-in HTML elements. For example:

```
<MyComponent>Hello world!</MyComponent>
```

This is valid JSX, and `props.children` in `MyComponent` will simply be the string `"Hello world!"`. HTML is unescaped, so you can generally write JSX just like you would write HTML in this way:

```
<div>This is valid HTML &amp;amp; JSX at the same time.</div>
```

JSX removes whitespace at the beginning and ending of a line. It also removes blank lines. New lines adjacent to tags are removed; new lines that occur in the middle of string literals are condensed into a single space. So these all render to the same thing:

```
<div>Hello World</div>

<div>
  Hello World
</div>

<div>
  Hello
  World
</div>

<div>

  Hello World
</div>
```

JSX Children

You can provide more JSX elements as the children. This is useful for displaying nested components:

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

You can mix together different types of children, so you can use string literals together with JSX children. This is another way in which JSX is like HTML, so that this is both valid JSX and valid HTML:

```
<div>
  Here is a list:
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

A React component can also return an array of elements:

```
render() {
  // No need to wrap list items in an extra element!
  return [
    // Don't forget the keys :)
    <li key="A">First item</li>,
    <li key="B">Second item</li>,
    <li key="C">Third item</li>,
  ];
}
```

JavaScript Expressions as Children

You can pass any JavaScript expression as children, by enclosing it within `{}`. For example, these expressions are equivalent:

```
<MyComponent>foo</MyComponent>

<MyComponent>{'foo'}</MyComponent>
```

This is often useful for rendering a list of JSX expressions of arbitrary length. For example, this renders an HTML list:

```
function Item(props) {
  return <li>{props.message}</li>;
}

function TodoList() {
  const todos = ['finish doc', 'submit pr', 'nag dan to review'];
  return (
    <ul>
      {todos.map((message) => <Item key={message} message={message} />)}
    </ul>
  );
}
```

JavaScript expressions can be mixed with other types of children. This is often useful in lieu of string templates:

```
function Hello(props) {
  return <div>Hello {props.addressee}!</div>;
}
```

Functions as Children

Normally, JavaScript expressions inserted in JSX will evaluate to a string, a React element, or a list of those things. However, `props.children` works just like any other prop in that it can pass any sort of data, not just the sorts that React knows how to render. For example, if you have a custom component, you could have it take a callback as `props.children`:

```
// Calls the children callback numTimes to produce a repeated component
function Repeat(props) {
  let items = [];
  for (let i = 0; i < props.numTimes; i++) {
    items.push(props.children(i));
  }
  return <div>{items}</div>;
}

function ListOfTenThings() {
  return (
    <Repeat numTimes={10}>
      {(index) => <div key={index}>This is item {index} in the list</div>}
    </Repeat>
  );
}
```

Children passed to a custom component can be anything, as long as that component transforms them into something React can understand before rendering. This usage is not common, but it works if you want to stretch what JSX is capable of.

Booleans, Null, and Undefined Are Ignored

`false`, `null`, `undefined`, and `true` are valid children. They simply don't render. These JSX expressions will all render to the same thing:

```
<div />

<div></div>

<div>{false}</div>

<div>{null}</div>

<div>{undefined}</div>

<div>{true}</div>
```

This can be useful to conditionally render React elements. This JSX only renders a `<Header />` if `showHeader` is `true`:

```
<div>
  {showHeader && <Header />}
  <Content />
</div>
```

One caveat is that some "falsy" values, such as the `0` number, are still rendered by React. For example, this code will not behave as you might expect because `0` will be printed when `props.messages` is an empty array:

```
<div>
  {props.messages.length &&
    <MessageList messages={props.messages} />
  }
</div>
```

To fix this, make sure that the expression before `&&` is always boolean:

```
<div>
  {props.messages.length > 0 &&
    <MessageList messages={props.messages} />
  }
</div>
```

Conversely, if you want a value like `false`, `true`, `null`, or `undefined` to appear in the output, you have to convert it to a string first:

```
<div>
  My JavaScript variable is {String(myVariable)}.
</div>
```