Chapter 21: JSX

Section 21.1: Props in JSX

There are several different ways to specify props in JSX.

JavaScript Expressions

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

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

Inside the `MyComponent`, the value of `props.count` 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.

String Literals

Of course, you can just pass any `string literal` as a `prop` too. 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 Value

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

```
<MyTextBox autocomplete />

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

However, the React team says in their docs **using this approach is not recommended**, because it can be confused with the ES6 object shorthand `{foo}` which is short for `{foo: foo}` rather than `{foo: true}`. They say 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 Case1() {
  return <Greeting firstName="Kaloyab" lastName="Kosev" />;
}
```

```
function Case2() {
  const person = {firstName: 'Kaloyan', lastName: 'Kosev'};
  return <Greeting {...person} />;
}
```

Section 21.2: 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>
    <h1>Hello world!</h1>
</MyComponent>
```

This is valid JSX, and `props.children` in MyComponent will simply be `<h1>Hello world!</h1>`.

Note that **the HTML is unescaped**, so you can generally write JSX just like you would write HTML.

Bare in mind, that in this case JSX:

- removes whitespace at the beginning and ending of a line;
- 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.

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>
  <h2>Here is a list</h2>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

Note that a React component **can't return multiple React elements, but a single JSX expression can have multiple children**. So if you want a component to render multiple things you can wrap them in a `div` like the example above.

JavaScript Expressions

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:

```
const Item = ({ message }) => (
  <li>{ message }</li>
);

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

Note that JavaScript expressions can be mixed with other types of children.

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`:

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

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

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.

Ignored Values

Note that **`false`**, **`null`**, **`undefined`**, and **`true`** are valid children. But they simply don't render. These JSX expressions

will all render to the same thing:

```
<MyComponent />

<MyComponent></MyComponent>

<MyComponent>{false}</MyComponent>

<MyComponent>{null}</MyComponent>

<MyComponent>{true}</MyComponent>
```

This is extremely useful to conditionally render React elements. This JSX only renders a if `showHeader` is true:

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

One important 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>
```

One approach to fix this is to make sure that the expression before the `&&` is always boolean:

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

Lastly, bare in mind that 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>
```