Types of Children

In JSX expressions, the content between an opening and closing tag is passed as a unique prop called children. There are several ways to pass children, such as rendering string literals or using JSX elements and JavaScript expressions. It is also essential to understand the types of JavaScript values that are ignored as children and don't render anything. Let's explore these in a bit more detail:

String literals

String literals refer to simple JavaScript strings. They can be put between the opening and closing tags, and the `children` prop will be that string.

```
<MyComponent>Little Lemon</MyComponent>
```

In the above example, the `children` prop in `MyComponent` will be simply the string "Little Lemon".

There are also some rules JSX follows regarding whitespaces and blank lines you need to bear in mind, so that you understand what to expect on your screen when those edge cases occur.

1. JSX removes whitespaces at the beginning and end of a line, as well as blank lines:

```
<div>    Little Lemon   </div>
 <div>
   Little Lemon
 </div>
```

2. New lines adjacent to tags are removed:

```
<div>

  Little Lemon
</div>
```

3. JSX condenses new lines that happen in the middle of string literals into a single space:

```
<div>
  Little
  Lemon
</div>
```

That means that all the instances above render the same thing.

JSX Elements

You can provide JSX elements as children to display nested components:

```
<Alert>
  <Title />
  <Body />
```

```
  </Alert>
```

JSX also enables mixing and matching different types of children, like a combination of string literals and JSX elements:

```
<Alert>
  <div>Are you sure?</div>
  <Body />
</Alert>
```

A React component can also return a bunch of elements without wrapping them in an extra tag. For that, you can use React Fragments either using the explicit component imported from React or empty tags, which is a shorter syntax for a fragment. A React Fragment component lets you group a list of children without adding extra nodes to the DOM. You can learn more about fragments in the additional resources unit from this lesson.

The two code examples below are equivalent, and it's up to your personal preference what to choose, depending on whether you prefer explicitness or a shorter syntax:

```
return (
  <React.Fragment>
    <li>Pizza margarita</li>
    <li>Pizza diavola</li>
  </React.Fragment>
);

return (
  <>
    <li>Pizza margarita</li>
    <li>Pizza diavola</li>
  </>
);
```

JavaScript Expressions

You can pass any JavaScript expression as children by enclosing it within curly braces, `{}`. The below expressions are identical:

`<MyComponent>Little Lemon</MyComponent>`

`<MyComponent>{'Little Lemon'}</MyComponent>`

This example is just for illustration purposes. When dealing with string literals as children, the first expression is preferred.

Earlier in the course, you learned about lists. JavaScript expressions can be helpful when rendering a list of JSX elements of arbitrary length:

```
function Dessert(props) {
  return <li>{props.title}</li>;
}

function List() {
  const desserts = ['tiramisu', 'ice cream', 'cake'];
  return (
    <ul>
```

```
        {desserts.map((dessert) => <Item key={dessert} title={dessert} />)}
      </ul>
    );
  }
```

Also, you can mix JavaScript expressions with other types of children without having to resort to string templates, like in the example below:

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

Functions

Suppose you insert a JavaScript expression inside JSX. In that case, React will evaluate it to a string, a React element, or a combination of the two. However, the children prop works just like any other prop, meaning it can be used to pass any type of data, like functions.

Function as children is a React pattern used to abstract shared functionality that you will see in detail in the next lesson.

Booleans, Null and Undefined, are ignored

false, null, undefined, and true are all valid children. They simply don't render anything. The below expressions will all render the same thing:

`<div />`

`<div></div>`

`<div>{false}</div>`

`<div>{null}</div>`

`<div>{undefined}</div>`

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

Again, this is all for demonstration purposes so that you know what to expect on your screen when these special values are used in your JSX.

When used in isolation, they don't offer any value. However, boolean values like true and false can be useful to conditionally render React elements, like rendering a Modal component only if the variable `showModal` is true

```
<div>
  {showModal && <Modal />}
</div>
```

However, keep in mind that React still renders some "false" values, like the 0 number. For example, the below code will not behave as you may expect because 0 will be printed when `props.desserts` is an empty array:

```
<div>
  {props.desserts.length &&
    <DessertList desserts={props.desserts} />
  }
</div>
```

To fix this, you need to make sure the expression before `&&` is always boolean:

```
<div>
  {props.desserts.length > 0 &&
    <DessertList desserts={props.desserts} />
  }
</div>

<div>
  {!!props.desserts.length &&
    <DessertList desserts={props.desserts} />
  }
</div>
```

Conclusion

You have learned about different types of children in JSX, such as how to render string literals as children, how JSX elements and JavaScript expressions can be used as children, and the boolean, null or undefined values that are ignored as children and don't render anything.