Dissecting props

Recall that much like parameters in a JavaScript function which allow you to pass in values as arguments, React uses properties, or **props**, to pass data between components. But how exactly do they work?

In this reading, you'll use a transpiler to break JSX code to plain JavaScript, making its purpose more understandable.

Remember first that JSX code in React is just syntactic sugar - meaning, a nicer way to write some hard-to-read code.

For the browser to understand this syntactic sugar, you need to transpile JSX down to plain JavaScript code. You have a resource online, at the URL of babeljs.io, which allows you to inspect the results of this transpiling. Once you visit the website, make sure to navigate to the *Try it out* link in the main navigation.

For example, let's say you have a component that returns a piece of JSX:

```
function App() {
  return <h1>Hello there</h1>
}
```

… if you used the Babel transpiler to transpile this JSX syntactic sugar code down to plain JavaScript code, you'd get back some unusual code:

```
"use strict";
function App() {
    return /*#__PURE__*/React.createElement("h1", null, "Hello there");
}
```

You just want to focus on the `React.createElement("h1", null, "Hello there");` part. You can ignore the rest.

This means that the `createElement` function receives three arguments:

1. The wrapping element to render.
2. A null value (which is there to show an absence of an expected JavaScript object value).
3. The inner content that will go inside the wrapping element.

Interestingly, the inner content that will go inside the wrapping element can also be a call to the `createElement` function.

For example, let's say you have a slightly more complex JSX element structure:

```
function App() {
  return (
```

```
    <div>
    <h1>Hello there</h1>
    </div>
  )
}
```

… the transpiled return statement in plain JavaScript again returns two `createElement` functions:

```
"use strict";
function App() {
  return /*#__PURE__*/React.createElement("div", null, /*#__PURE__*/React.createElement("h1", null, "
}
```

If you format this output, remove the `"use strict"` line, and remove the `__PURE__` comments, you get a more readable output:

```
function App() {
  return React.createElement(
    "div",
    null,
    React.createElement("h1", null, "Hello there")
  );
}
```

So now the third argument of the outer-most `React.createElement` call is another `React.createElement` call.

This is how you can nest as many elements as you want.

This means that a nested JSX structure is just a bunch of nested `React.createElement` calls, passed in to other `React.createElement` calls as their third argument.

The second – `null` – argument

The second argument of `null` can – in this case – be replaced with an empty object.

In that case, your code would contain a pair of curly braces instead of the word `null`:

```
"use strict";

function App() {
  return React.createElement(
    "div",
    {},
    React.createElement("h1", {}, "Hello there")
  );
}
```

This object is referred to as the *props* object. It is the main mechanism of sending data from a parent component to a child component in React.

The way this works is described in React docs using the following code:

```
React.createElement(
  type,
  [props],
  [...children]
)
```

The third argument (...children)

This is the inner content that will go inside the wrapping element. It's what makes it possible to nest elements inside other elements, mimicking the way that HTML works.

In this reading you've learned how to use a transpiler to break JSX code to plain JavaScript, making its purpose more understandable.