Module 6

Transferring Properties (Props)

6. Transferring Properties (Props)

There is a frustrating side to working with properties. We kinda saw this side in the previous chapter. Passing properties from one component to another is nice and simple when you are dealing with only one layer of components. When you wish to send a property across multiple layers of components, things start getting complicated.

Things getting complicated is never a good thing, so in this chapter, let's see what we can do to make working with properties across multiple layers of components easy.

Problem Overview

Let's say that you have a deeply nested component, and its hierarchy (modeled as awesomely colored circles) looks like Figure 6-1.

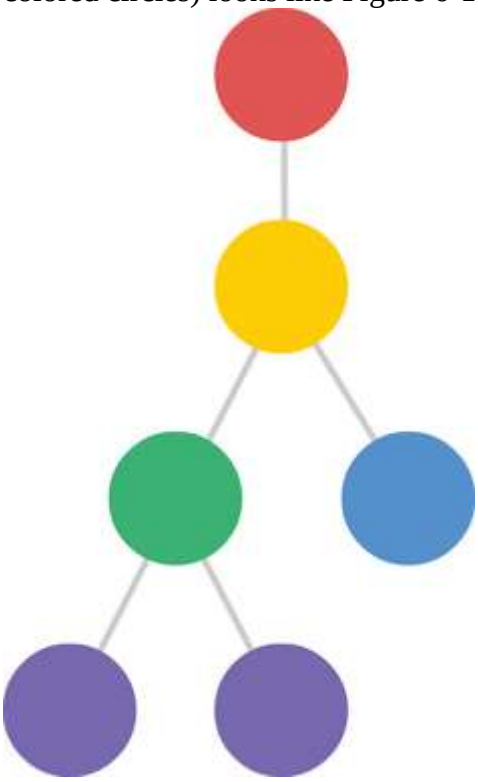

Figure 6-1 The component hierarchy.

What you want to do is pass a property from your red circle all the way down to our purple circles where it will be used. What we can't do is the very obvious and straightforward thing shown in Figure 6-2.

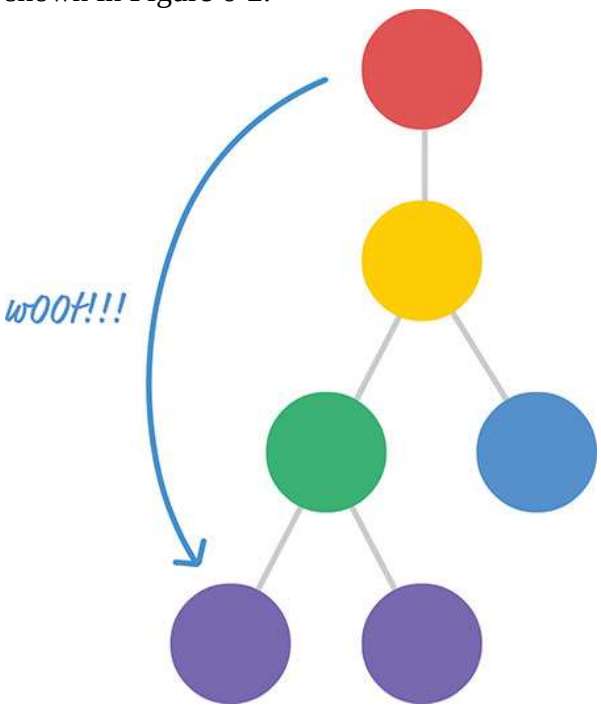

Figure 6-2 Can't do this.

You can't pass a property directly to the component or components that you wish to target. The reason has to do with how React works. **React enforces a chain of command where properties have to flow down from a parent component to an immediate child component.** This means you can't skip a layer of children when sending a property. This also means your children can't send a property back up to a parent. **All communication is one-way from the parent to the child.**
Under these guidelines, passing a property from our red circle to our purple circle looks a little bit like Figure 6-3.

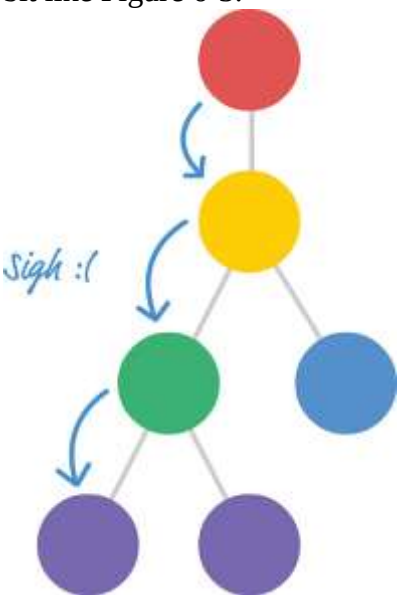

Figure 6-3 The property is passed from parent to child.

Every component that lies on the intended path has to receive the property from its parent and then re-send that property to its child. This process repeats until your property reaches its intended destination. The problem is in this receiving and re-sending step.
If we had to send a property called color from the component representing our red circle to the component representing our purple circle, its path to the destination would look something like Figure 6-4.

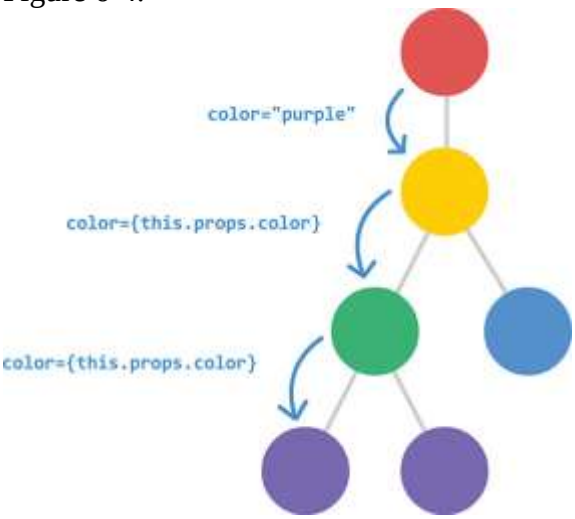

Figure 6-4 Sending the color property.

Now, imagine we have two properties that we need to send, as in Figure 6-5.

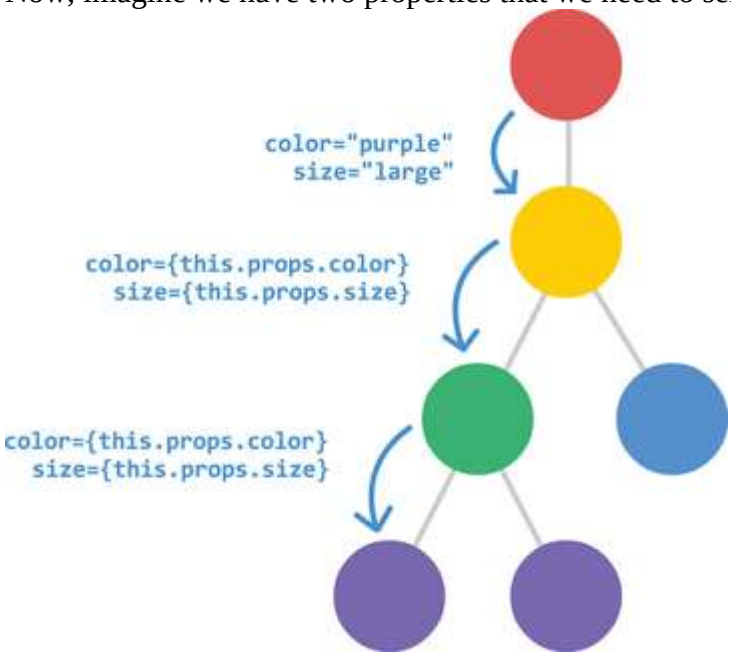

Figure 6-5 Sending two properties.

What if we wanted to send three properties? Or four?

We can quickly see that this approach is neither scalable nor maintainable. For every additional property we need to communicate, we are going to have to add an entry for it as part of declaring each component. If we decide to rename our properties at some point, we will have to ensure that every instance of that property is renamed as well. If we remove a property, we need to remove the property from every component that relied on it. Overall, these are the kinds of situations we try to avoid when writing code. What can we do about this?

Detailed Look at the Problem

In the previous section, we talked at a high level about what the problem is. Before we can dive into figuring out a solution, we need to go beyond diagrams and look at a more detailed example with real code. We need to take a look at something like the following:

```
var Display = React.createClass({
  render: function() {
    return(
      <div>
        <p>{this.props.color}</p>
        <p>{this.props.num}</p>
        <p>{this.props.size}</p>
      </div>
    );
  }
});

var Label = React.createClass({
  render: function() {
    return (
      <Display color={this.props.color}
            num={this.props.num}
            size={this.props.size}/>
    );
  }
});

var Shirt = React.createClass({
  render: function() {
      return (
        <div>
          <Label color={this.props.color}
                num={this.props.num}
                size={this.props.size}/>
        </div>
      );
    }
});

ReactDOM.render(
  <div>
    <Shirt color="steelblue" num="3.14" size="medium"/>
  </div>,
  document.querySelector("#container")
);
```

Take a few moments to understand what is going on. Once you have done that, let's walk through this example together.
What we have is a Shirt component that relies on the output of the Label component which relies on the output of the Display component. (Try saying that sentence five time fast!) Anyway, the component hierarchy can be seen in Figure 6-6.

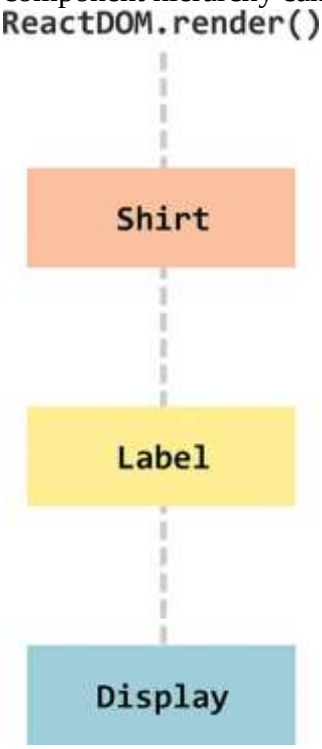

Figure 6-6 The component hierarchy.

When you run this code, what gets output is nothing special. It is just three lines of text (see Figure 6-7).

Figure 6-7 The three lines of text.

The interesting part is how the text gets there. Each of the three lines of text that you see maps to a property we specified at the very beginning inside ReactDOM.render:

```
<Shirt color="steelblue" num="3.14" size="medium"/>
```

The color, num, and size properties (and their values) make a journey all the way to the Display component that would make even the most seasoned world traveler jealous. Let's follow these properties from their inception to when they get consumed, and I do realize that a lot of this will be a review of what you've already seen. If you find yourself getting bored, feel free to skip on to the next section. With that said, onwards and upwards!

Life for our properties starts inside ReactDOM.render when our Shirt component gets called with the color, num, and size properties specified:

```
ReactDOM.render(
  <div>
    <Shirt color="steelblue" num="3.14" size="medium"/>
  </div>,
  document.querySelector("#container")
);
```

We not only define the properties, we also initialize them with the values they will carry. Inside the Shirt component, these properties are stored inside the props object. To transfer these properties on, we need to explicitly access these properties from the props object and list them as part of the component call. The following is an example of what that looks like when our Shirt component calls our Label component:

```
var Shirt = React.createClass({
  render: function() {
      return (
        <div>
          <Label color={this.props.color}
                 num={this.props.num}
                 size={this.props.size}/>
        </div>
      );
    }
});
```

Notice that the color, num, and size properties are listed again. The only difference from what we saw with the ReactDOM.render call is that the values for each property are taken from their respective entry in the props object as opposed to being manually entered.

When our Label component goes live, it has its props object properly filled out with the color, num, and size properties stored. You can probably see a pattern forming here.

The Label component continues the tradition by repeating the same steps and calling the Display component:

```
var Label = React.createClass({
  render: function() {
    return (
      <Display color={this.props.color}
               num={this.props.num}
               size={this.props.size}/>
    );
  }
});
```

Phew. All we wanted to do was have our Display component display some values for color, num, and size. The only complication was that the values we wanted to display were originally defined as part of ReactDOM.render. The annoying solution is the one you see here where every component along the path to the destination needs to access and re-define each property as part of passing it along. That's just terrible. We can do better than this, and you will see how in a few moments!

Meet the Spread Operator

The solution to all of our problems lies in something new to JavaScript known as the spread operator. What the spread operator does is a bit bizarre to explain without some context, so I'll first give you an example and then bore you with a definition.
Take a look at the following snippet:

```
var items = ["1", "2", "3"];

function printStuff(a, b, c) {
  console.log("Printing: " + a + " " + b + " " + c);
}
```

We have an array called items that contains three values. We also have a function called printStuff that takes three arguments. What we want to do is specify the three values from our items array as arguments to the printStuff function. Sounds simple enough, right?
Here is one really common way of doing that:

```
printStuff(items[0], items[1], items[2]);
```

We access each array item individually and pass it in to our printStuff function. With the spread operator, we now have an easier way. You don't have to specify each item in the array individually at all. You can just do something like this:

```
printStuff(...items);
```

The spread operator is the ... characters before our items array, and using ...items is identical to listing items[0], items[1], and items[2] individually like we did earlier. The printStuff function will run and print the numbers 1, 2, and 3 to our console. Pretty cool, right?

Now that you've seen the spread operator in action, it's time to define it. **The spread operator enables you to unwrap an array into its individual elements.** The spread operator does a few more things as well, but that's not important for now. We are going to only use this particular side of the spread operator to solve our property transferring problem!

Properly Transferring Properties

We just saw an example where we used the spread operator to avoid having to enumerate every single item in our array as part of passing it to a function:

```
var items = ["1", "2", "3"];

function printStuff(a, b, c) {
  console.log("Printing: " + a + " " + b + " " + c);
}

// using the spread operator
printStuff(...items);

// without using the spread operator
printStuff(items[0], items[1], items[2]);
```

The situation we are facing with transferring properties across components is very similar to our problem of accessing each array item individually. Allow me to elaborate.
Inside a component, our props object looks as follows:

```
var props = {
  color: "steelblue",
  num: "3.14",
  size: "medium"
}
```

As part of passing these property values to a child component, we manually access each item from our props object:

```
<Display color={this.props.color}
      num={this.props.num}
      size={this.props.size}/>
```

Wouldn't it be great if there was a way to unwrap an object and pass on the property/value pairs just like we were able to unwrap an array using the spread operator?

As it turns out, there is a way. It actually involves the spread operator as well. I'll explain how later, but what this means is that **we can call our Display component by using ...props:**

```
<Display {...props}/>
```

By using ...props, the runtime behavior is the same as specifying the color, num, and size properties manually. This means our earlier example can be simplified as follows (pay attention to the **highlighted** lines):

```
var Display = React.createClass({
  render: function() {
    return(
      <div>
        <p>{this.props.color}</p>
        <p>{this.props.num}</p>
        <p>{this.props.size}</p>
      </div>
    );
  }
});

var Label = React.createClass({
  render: function() {
    return (
      <Display {...this.props}/>
    );
  }
});

var Shirt = React.createClass({
  render: function() {
      return (
        <div>
          <Label {...this.props}/>
        </div>
      );
    }
});

ReactDOM.render(
  <div>
    <Shirt color="steelblue" num="3.14" size="medium"/>
  </div>,
  document.querySelector("#container")
);
```

If you run this code, the end result is going to be unchanged from what we had earlier. The biggest difference is that we are no longer passing in expanded forms of each property as part of calling each component. This solves all the problems we originally set out to solve.

By using the spread operator, if you ever decide to add properties, rename properties, remove properties, or do any other sort of property-related shenanigans, you don't have to make a billion different changes. You make one change at the spot where you define your property. You make another change at the spot you consume the property. That's it. All of the intermediate components that merely transfer the properties on will remain untouched, for the {...this.props} expression contains no details of what goes on inside it.

Conclusion

As designed by the ES6/ES2015 committee, the spread operator is designed to work only on arrays and array-like creatures (aka that which has a Symbol.iterator property). The fact that it works on object literals like our props object is due to React extending the standard. As of now, no browser currently supports using the spread object on object literals. The reason our example works is because of Babel. Besides turning all of our JSX into something our browser understands, Babel also turns cutting-edge and experimental features into something cross-browser friendly. That is why we are able to get away with using the spread operator on an object literal, and that is why we are able to elegantly solve the problem of transferring properties across multiple layers of components!