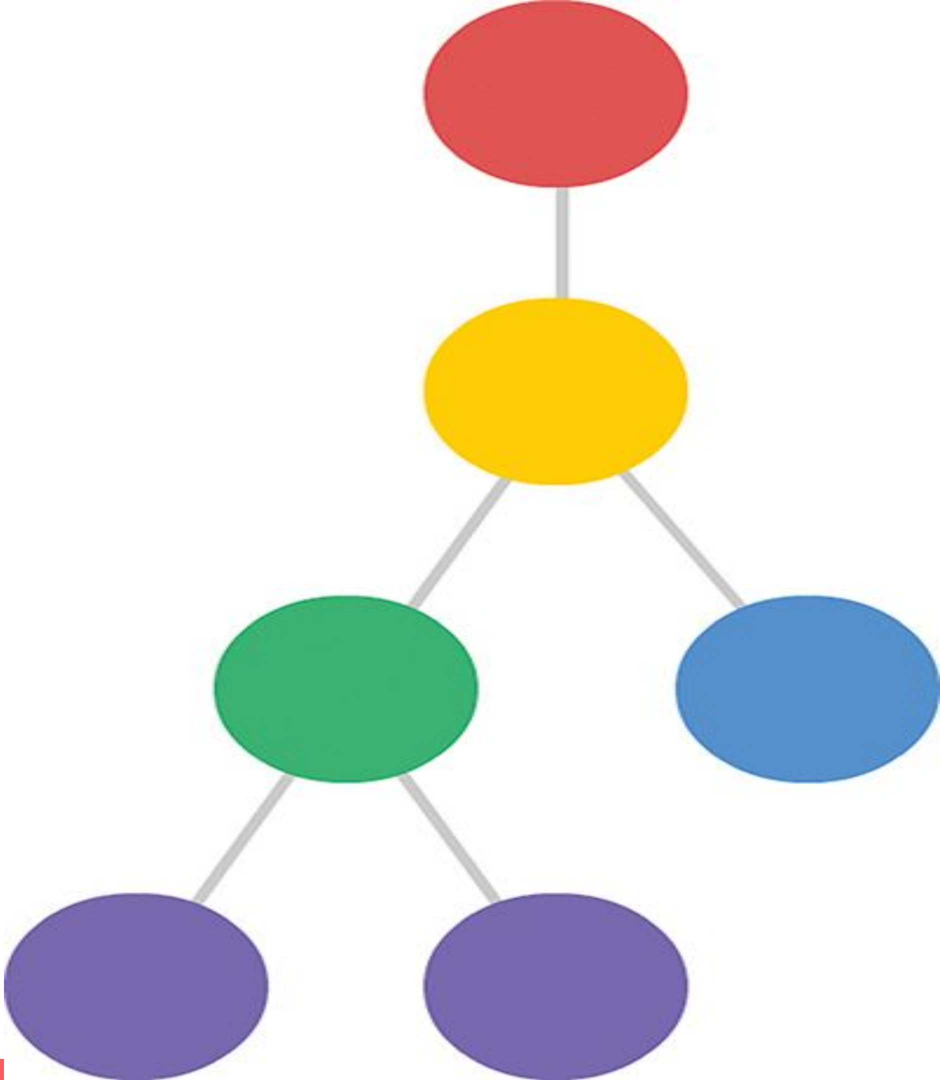
Transferring Properties

CLASS 5

Problem Overview

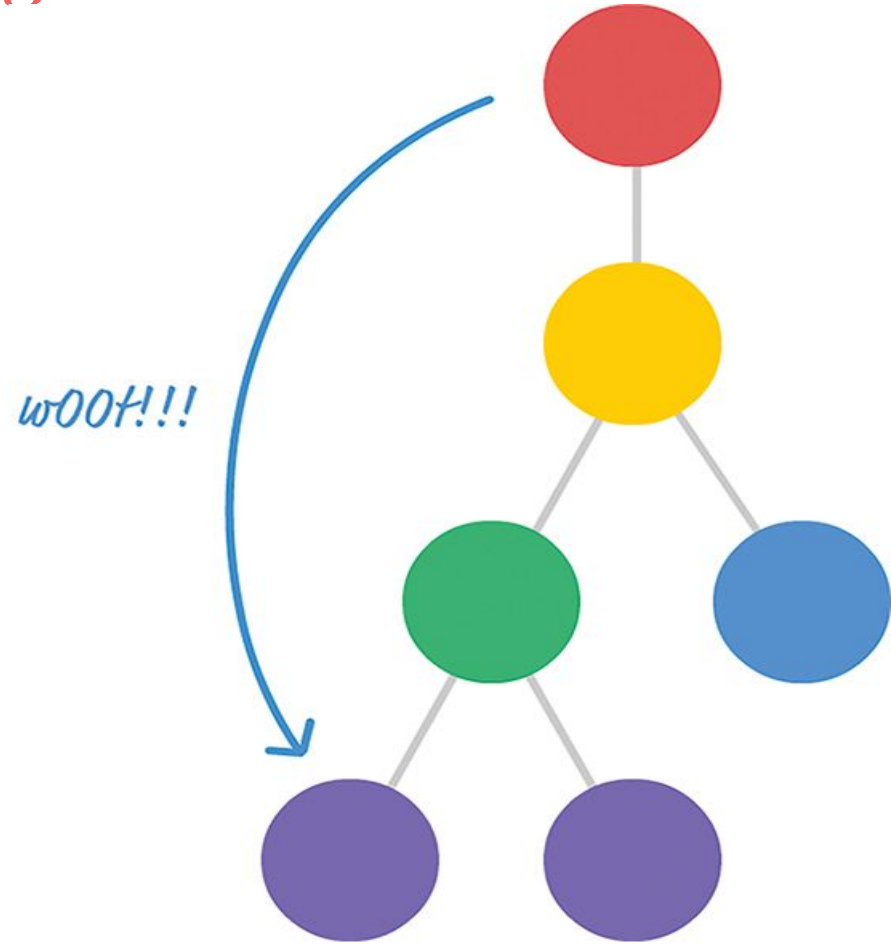
In a deeply nested component, and its hierarchy(modeled as awesomely colored circles).Transferring property from red circle all the way down to the purple circles, where it will be used.



Properties can not pass Straightforward

You can’t pass a property directly to the component or components that you want to target. The reason has to do with how React works.

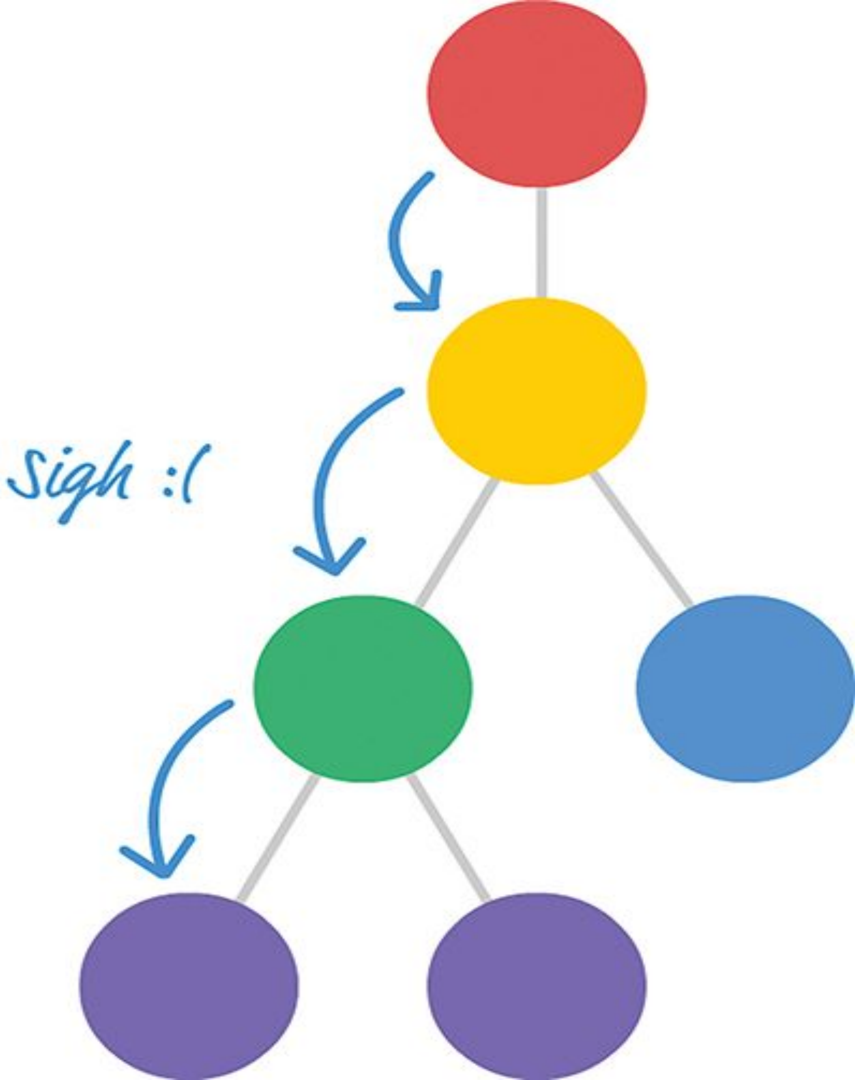
React enforces a chain of command in which properties have to flow down from a parent component to an immediate child component. This



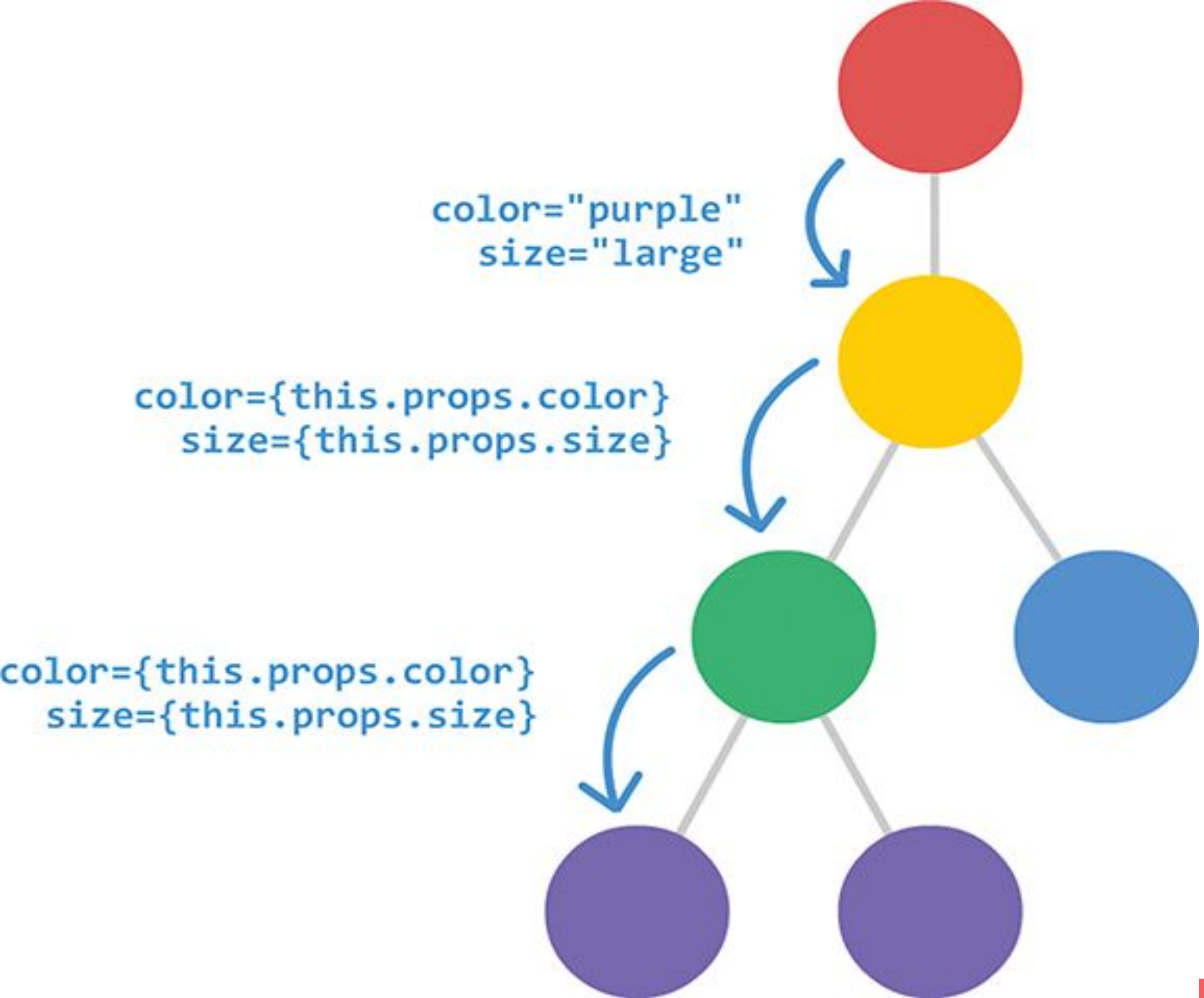
How React Work

React enforces a chain of command in which properties have to flow down from a parent component to an immediate child component.

Every component that lies on the intended path has to receive the property from its parent and then resend 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.



Sending two properties(Color And Size)

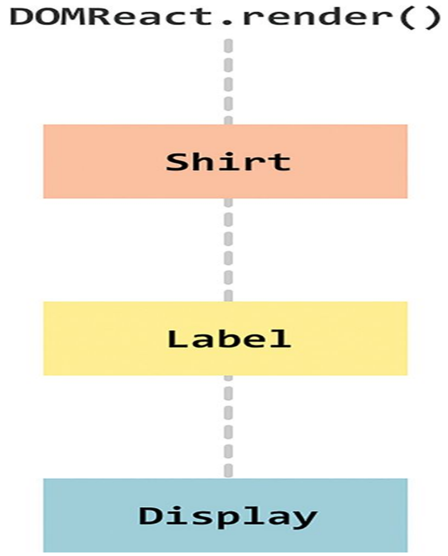


Problem with the hierarchy approach

This approach is neither scalable nor maintainable. For every additional property we need to communicate, we have to add an entry for it as part of declaring each component.

component. If we decide to rename our properties at some point, we 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 being used across every component that relied on it.

We have a Shirt component that relies on the output of the Label component, which relies on the output of the Display component. shows the component hierarchy.



```
class Display extends
React.Component {
render() {
return (
<div>
<p>{this.props.color}</p>
<p>{this.props.num}</p>
<p>{this.props.size}</p>
</div>
);
}
}
```

```
class Label extends
React.Component {
render() {
return (
<Display
color={this.props.color}
num={this.props.num}
size={this.props.size}/>
);
}
}
```

```
class Shirt extends
React.Component {
render() {
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") );
```

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, but 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.
- Shirt component calls our Label component
- The Label component continues the tradition by repeating the same steps and calling the Display component
- The Display component call contains the same listing of properties and their values taken from our Label component's props object. The only good news from all this is that we're almost done here.The Display component just displays the properties as they were populated inside its props object:
- *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, with every component along the path to the destination needing to access and redefine each property as part of passing it along.*

Meet the Spread Operator

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. We want to specify the three values from our items array as arguments to the printStuff function.

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

With the spread operator, we now have an easier way. You don’t have to specify each item in the array individually; you can just do something like this:

printStuff(...items);

The spread operator is the ... characters before our items array

The spread operator allows you to unwrap an array into its individual elements.

A Better Way to Transfer Properties

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}/>
```

we can call our Display component by using ...this.props:

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

```
class Display extends
React.Component {

render() {

return (

<div>

<p>{this.props.color}</p>

<p>{this.props.num}</p>

<p>{this.props.size}</p>

</div>

);

}

}
```

```
class Label extends
React.Component {
render() {
return (
<Display {...this.props} />
);
}
}
```

```
class Shirt extends React.Component {
render() {
return (
<div>
<Label {...this.props} />
</div>
);
}
}
```

If you run this code, the end result is 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 you define your property.

Conclusion

No browser currently supports using the spread object on object literals. Our example works because of Babel. Besides turning all our JSX into something our browser understands, Babel turns cutting-edge and experimental features into something that's friendly across browsers. That's why we're able to get away with using the spread operator on an object literal, and that's why we're able to elegantly solve the problem of transferring properties across multiple layers of components.

The important part to realize is that you can use the spread operator to transfer props from one component to another.

Is this the best way to transfer properties?

Meet JSX... Again!

JavaScript Extended

What Happens with JSX?

Relying on things like Babel to turn that JSX into

something the browsers understand: JavaScript.

JSX is for us only, when this JSX reaches the browser, it ends up getting turned into pure JavaScript:

Transpilation happens in background because of Babel, which is used to perform the JSX-to-JS transformation entirely in the browser.

This is JSX:

```
    <div style={cardStyle}>

    <Square color={this.props.color} />

    <Label color={this.props.color} />

    </div>
```

This is javascript:

```
    return React.createElement(

    "div",

    { style: cardStyle },

    React.createElement(Square, { color:
    this.props.color }),

    React.createElement(Label, { color:
    this.props.color })

    );
```

JSX Quirks to Remember : Evaluating Expressions

JSX is treated like JavaScript. It is not limited to deal with static content like:

```
class Stuff extends React.Component {

render() {

return (

<h1>Boring static content!</h1>

);

}

};
```

The values return can be dynamically generated, just wrap the expression in curly braces:

```
class Stuff extends React.Component {

render() {

return (

<h1>Boring {Math.random() * 100} content!</h1>

);

}

}
```

These curly braces allows first evaluate the expression and then return the result of the evaluation. Without them, expression returned as text: Boring Math.random() * 100 content!

JSX Quirks to Remember : Returning Multiple Elements

Returning multiple elements be like:

```
class Stuff extends React.Component {

render() {

return (

[

<p>I am</p>,

<p>returning a list</p>,

<p>of things!</p>

]

);
```

On returning multiple items, it is not dealing with one detail, which depends on the version of React. So, need to specify a key attribute and a unique value for each item:

```
class Stuff extends React.Component {

render() {

return (

[

<p key="1">I am</p>,

<p key="2">returning a list</p>,

<p key="3">of things!</p>

]

);

}

}
```

How to know whether need a key attribute or not?
React will tell: *Warning: Each child in an array or iterator should have a unique "key" prop.*

JSX Quirks to Remember : Returning Multiple Elements

Another method of returning multiple elements: **Fragments**, like:

```
class Stuff extends React.Component {

        render() {

        return (

        <React.Fragment>

        <p>I am</p>

        <p>returning a list</p>

        <p>of things!</p>

        </React.Fragment>

        ); } }
```

Doesn't actually generate DOM element, but specifying key attribute is taken care of.

There can be a more condensed syntax instead of fully specifying React.Fragment… , which is just empty <> and </> tags:

```
        class Stuff extends React.Component {

        render() {

        return (

        <>

        <p>I am</p>

        <p>returning a list</p>

        <p>of things!</p>

        </>

        ); } }
```

JSX Quirks to Remember : You Can't Specify CSS Inline

The style attribute in JSX behaves differently from the style attribute in HTML. In HTML, CSS properties directly specified as values in style attribute:

```
<div style="font-family:Arial;font-size:24px">

<p>Blah!</p>

</div>
```

In JSX, the style attribute can't contain CSS inside it. Instead, it needs to refer to an object that contains styling information:

```
class Letter extends React.Component {

render() {

var letterStyle = {

padding: 10, margin: 10,  backgroundColor: this.props.bgcolor,

color: "#333", display: "inline-block", fontFamily: "monospace",

fontSize: "32", textAlign: "center"

};

return (

<div style={letterStyle}>

{this.props.children}

</div>

); } }
```

Object **letterStyle** that contains all the CSS properties (in camel-case JavaScript form) and their values.

JSX Quirks to Remember : Comments,

Specifying a comment as a child of a tag, then enclose comment within the { and } angle brackets to ensure that it is parsed as an expression:

```
ReactDOM.render(

<div className="slideIn">

<p className="emphasis">Gabagool!</p>

{/* I am a child comment */}

<Label/>

</div>,

document.querySelector("#container")

);
```

this case is a child of div element. If specify a comment wholly inside a tag, then specify single-line or multiline comment without having to use the { and } angle brackets:

```
ReactDOM.render(

<div className="slideIn">

<p className="emphasis">Gabagool!</p>

<Label

/* This comment goes across multiple lines */

className="colorCard" // end of line

/>

</div>,

document.querySelector("#container")

);
```

JSX Quirks to Remember : Capitalization, HTML Elements, and Components

Capitalization is important. To represent HTML elements, ensure that the HTML tag is lowercase:

```
ReactDOM.render(

<div>

<section>

<p>Something goes here!</p>

</section>

</div>,

document.querySelector("#container")

);
```

To represent components, the component name must be capitalized:

```
ReactDOM.render(

<div>

<MyCustomComponent/>

</div>,

document.querySelector("#container")

);
```

If the capitalization get wrong, React will not render content properly.

Your JSX Can Be Anywhere

In many situations, JSX won't be neatly arranged inside a render or return function, take a look at the following example:

```
var swatchComponent = <Swatch
color="#2F004F"></Swatch>;

ReactDOM.render(

<div>

{swatchComponent}

</div>,

document.querySelector("#container")

);
```

We have a variable called *swatchComponent* that is initialized to a line of JSX. When our *swatchComponent* variable is placed inside the render function, our Swatch component gets initialized. All of this is totally valid.

CONCLUSION:

It has been finally pieced together in one location the various bits of JSX information that the previous chapters introduced.

JSX is not HTML. It looks like HTML and behaves like it in many common scenarios, but it is ultimately designed to be translated into JavaScript.

Thank You