**Advanced React Techniques**

In this unit, you will learn a variety of useful techniques that React programmers are expected to know.

You’ll learn how to make a *stateless functional component*, how to make a *propType*, how to write a form, and how to use styles.

You’ll also be introduced to your second programming pattern: dividing components into *presentational components* and *container components*.

**Inline Styles**

There are many different ways to use styles in React. This lesson is focused on one of them: *inline styles.*

An inline style is a style that’s written as an *attribute*, like this:

<h1 style={{ color: 'red' }}>Hello world</h1>

Notice the double curly braces. What are those for?

The *outer* curly braces inject JavaScript into JSX. They say, “everything between us should be read as JavaScript, not JSX.”

The *inner* curly braces create a JavaScript object literal. They make this a valid JavaScript object:

{ color: 'red' }

If you inject an object literal into JSX, and your entire injection is *only* that object literal, then you will end up with double curly braces. There’s nothing unusual about how they work, but they look funny and can be confusing.

**Instructions**

**1.**

Click Run to see the <h1></h1> rendered to the browser. How can you add styles to this poor <h1></h1>?

Checkpoint 2 Passed

**2.**

Let’s give the <h1></h1> an *inline style*.

Give the <h1></h1> an attribute with a *name* of style. The attribute’s *value* should evaluate to this object:

{ background: 'lightblue', color: 'darkred' }

import React from 'react';

import ReactDOM from 'react-dom';

const styleMe = <h1 style={{ background: 'lightblue', color: 'darkred' }}>Please style me! I am so bland!</h1>;

ReactDOM.render(

  styleMe,

  document.getElementById('app')

);

**Make A Style Object Variable**

That’s all that you need to apply basic styles in React! Simple and straightforward.

One problem with this approach is that it becomes obnoxious if you want to use more than just a few styles. An alternative that’s often nicer is to store a style object in a *variable*, and then inject that variable into JSX.

Look in the code editor for an example. The style object is *defined* on lines 3-6, and then *injected* on line 11.

If you aren’t used to using modules, then this code may have made you twitch uncontrollably:

const style = {

color: 'darkcyan',

background: 'mintcream'

};

Defining a variable named style in the top-level scope would be an extremely bad idea in many JavaScript environments! In React, however, it’s totally fine.

Remember that every file is invisible to every other file, except for what you choose to expose via module.exports. You could have 100 different files, all with global variables named style, and there could be no conflicts.

**Instructions**

**1.**

Select **styleMe.js**. Make a new line after import ReactDOM from 'react-dom';.

On this new line, declare a new constant named styles. Set styles equal to this object:

{

background: 'lightblue',

color: 'darkred'

}

Checkpoint 2 Passed

**2.**

Change the *value* of your <h1></h1>‘s style attribute. Make style‘s value equal to your new styles variable.

Since you aren’t injecting an object *literal* anymore, you will no longer need to use double curly braces.

# Style Name Syntax

In regular JavaScript, style names are written in hyphenated-lowercase:

const styles = {

'margin-top': "20px",

'background-color': "green"

};

In React, those same names are instead written in camelCase:

const styles = {

marginTop: "20px",

backgroundColor: "green"

};

This has zero effect on style property values, only on style property names.

import React from 'react';

import ReactDOM from 'react-dom';

const styles = {

  background: 'lightblue',

  color:      'darkred',

  marginTop: '100px',

  fontSize: '50px'

};

const styleMe = <h1 style={styles}>Please style me! I am so bland!</h1>;

ReactDOM.render(

  styleMe,

  document.getElementById('app')

);

**Style Value Syntax**

In the last exercise, you learned how style *names* are slightly different in React than they are in regular JavaScript.

In this exercise, you will learn how style *values* are slightly different in React than they are in regular JavaScript.

In regular JS, style *values* are almost always strings. Even if a style value is numeric, you usually have to write it as a string so that you can specify a unit. For example, you have to write "450px" or "20%".

In React, if you write a style value as a *number*, then the unit "px" is assumed.

How convenient! If you want a font size of 30px, you can write:

{ fontSize: 30 }

If you want to use units other than “px,” you can use a string:

{ fontSize: "2em" }

Specifying “px” with a string will still work, although it’s redundant.

A few specific styles will *not* automatically fill in the “px” for you. These are styles where you aren’t likely to use “px” anyway, so you don’t really have to worry about it. [Here is a list of styles that don’t assume “px”.](https://facebook.github.io/react/tips/style-props-value-px.html)

**Instructions**

**1.**

In your styles object, change any property values that end in “px” from strings into numbers.

# Separate Container Components From Presentational Components: Explanation

In this lesson, you will learn your second programming pattern: separating presentational components from display components.

Click Run. In the browser, navigate to https://localhost:8000.

You are looking at an rendered <GuineaPigs /> component.

<GuineaPigs />‘s job is to render a photo carousel of guinea pigs. It does this perfectly well! And yet, it has a problem: it does too much stuff.

We can break <GuineaPigs /> into smaller components, but before we do: how do we know that GuineaPigs does too much stuff? How can you tell when a component has too many responsibilities?

Separating container components from presentational components helps to answer that question. It shows you when it might be a good time to divide a component into smaller components. It also shows you how to perform that division.