APPENDIX A
ES6 Syntax

Some of the code samples in this book use what's known as ES6 syntax. If you're not familiar with ES6 syntax, don't worry — it's a pretty straightforward translation from the JavaScript you might be accustomed to.

ES6 refers to ECMAScript 6, also known as "Harmony," the forthcoming version of ECMAScript. JavaScript is an implementation of ECMAScript. There's plenty of interesting history behind these naming conventions, but what you need to know is: ES6 is the "new" version of JavaScript, and extends the existing specification with some helpful new features.

React Native uses Babel, the JavaScript compiler, to transform our JavaScript and JSX code. One of Babel's features is its ability to compile ES6 syntax into ES5-compliant JavaScript, so we can use ES6 syntax throughout our React codebase.

Destructuring

Destructuring assignments provide us with a convenient shorthand for extracting data from objects.

Take this ES5-compliant snippet:

```
var myObj = {a: 1, b: 2};
var a = myObj.a;
var b = myObj.b;
```

We can use destructuring to do this more succinctly:

```
var {a, b} = {a: 1, b: 2};
```

You'll often see this used with `require` statements. When we `require` React, we're actually getting out an object. We *could* name components using the following syntax:

Example A-1. Importing the <View> component without destructuring

```
var React = require('react-native');
var View = React.View
```

But it's much nicer to use destructuring:

Example A-2. Using destructuring to import the <View> component

```
var { View } = require('react-native');
```

Importing Modules

Normally, we might use CommonJS module syntax to export our components and other JavaScript modules. In this system, we use `require` to import other modules, and assign a value to `module.exports` in order to make a file's contents available to other modules.

Example A-3. Requiring and exporting modules using CommonJS syntax

```
var OtherComponent = require('./other_component');

var MyComponent = React.createClass({
  ...
});

module.exports = MyComponent;
```

With ES6 module syntax, we can use the `export` and `import` commands instead. Here's the equivalent code, using ES6 module syntax:

Example A-4. Importing and exporting modules using ES6 module syntax

```
import OtherComponent from './other_component';

var MyComponent = React.createClass({
  ...
});

export default MyComponent;
```

Function Shorthand

ES6's function shorthand is also convenient. In ES5-compliant JavaScript, we define functions like so:

Example A-5. Longhand function declaration

```
render: function() {
  return <Text>Hi</Text>;
}
```

Writing out `function` over and over again can get annoying. Here's the same function, this time applying ES6's function shorthand:

Example A-6. Shorthand function declaration

```
render() {
  return <Text>Hi</Text>;
}
```

Fat Arrow Functions

In ES5-compliant JavaScript, we often need to `bind` our functions to make sure that their context (i.e. the value of `this`) is as expected. This is especially common when dealing with callbacks.

Example A-7. Binding functions manually with ES5-compliant JavaScript

```
var callbackFunc = function(val) {
  console.log('Do something');
}.bind(this);
```

Fat arrow functions are automatically bound, so we don't need to do that ourselves.

Example A-8. Using a fat-arrow function for binding

```
var callbackFunc = (val) => {
  console.log('Do something');
};
```

String Interpolation

In ES5-compliant JavaScript, we might do something like this to build a string:

Example A-9. String concatenation in ES5-compliant JavaScript

```
var API_KEY = 'abcdefg';
var url = 'http://someapi.com/request&key=' + API_KEY;
```

ES6 provides us with tempate strings, which support multi-line strings and string interpolation. By enclosing a string in backticks, we can use insert other variable values using the `${}` syntax.

Example A-10. String interpolation in ES6

```
var API_KEY = 'abcdefg';
var url = `http://someapi.com/request&key=${API_KEY}`;
```