React uses ES6, and you should be familiar with some of the new features like:

- Classes
- Arrow Functions
- Variables (let, const, var)
- Array Methods like `.map()`
- Destructuring
- Modules
- Ternary Operator
- Spread Operator

React ES6 Classes

Classes

ES6 introduced classes.

A class is a type of function, but instead of using the keyword `function` to initiate it, we use the keyword `class`, and the properties are assigned inside a `constructor()` method.

Example

A simple class constructor:

```
class Car {
  constructor(name) {
    this.brand = name;
  }
}
```

Notice the case of the class name. We have begun the name, "Car", with an uppercase character. This is a standard naming convention for classes.

Now you can create objects using the Car class:

Example

Create an object called "mycar" based on the Car class:

```
class Car {
  constructor(name) {
    this.brand = name;
  }
}

const mycar = new Car("Ford");
```

Note: The constructor function is called automatically when the object is initialized.

Method in Classes

You can add your own methods in a class:

Example

Create a method named "present":

```
class Car {
  constructor(name) {
    this.brand = name;
  }

  present() {
    return 'I have a ' + this.brand;
  }
}

const mycar = new Car("Ford");
mycar.present();
```

As you can see in the example above, you call the method by referring to the object's method name followed by parentheses (parameters would go inside the parentheses).

Class Inheritance

To create a class inheritance, use the `extends` keyword.
A class created with a class inheritance inherits all the methods from another class:

Example

Create a class named "Model" which will inherit the methods from the "Car" class:

```
class Car {
  constructor(name) {
    this.brand = name;
  }

  present() {
    return 'I have a ' + this.brand;
  }
}

class Model extends Car {
  constructor(name, mod) {
    super(name);
    this.model = mod;
  }
  show() {
      return this.present() + ', it is a ' + this.model
  }
}
const mycar = new Model("Ford", "Mustang");
mycar.show();
```

The `super()` method refers to the parent class.
By calling the `super()` method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.
To learn more about classes, check out our JavaScript Classes section.

React ES6 Arrow Functions

Arrow Functions

Arrow functions allow us to write shorter function syntax:

Before:

```
hello = function() {
  return "Hello World!";
}
```

With Arrow Function:

```
hello = () => {
  return "Hello World!";
}
```

It gets shorter! If the function has only one statement, and the statement returns a value, you can remove the brackets *and* the `return` keyword:

Arrow Functions Return Value by Default:

```
hello = () => "Hello World!";
```

Note: This works only if the function has only one statement.

If you have parameters, you pass them inside the parentheses:

Arrow Function With Parameters:

```
hello = (val) => "Hello " + val;
```

In fact, if you have only one parameter, you can skip the parentheses as well:

Arrow Function Without Parentheses:

```
hello = val => "Hello " + val;
```

What About `this`?

The handling of `this` is also different in arrow functions compared to regular functions.
In short, with arrow functions there are no binding of `this`.
In regular functions the `this` keyword represented the object that called the function, which could be the window, the document, a button or whatever.
With arrow functions, the `this` keyword *always* represents the object that defined the arrow function.
Let us take a look at two examples to understand the difference.
Both examples call a method twice, first when the page loads, and once again when the user clicks a button.
The first example uses a regular function, and the second example uses an arrow function.
The result shows that the first example returns two different objects (window and button), and the second example returns the Header object twice.

Example

With a regular function, `this` represents the object that called the function:

```
class Header {
  constructor() {
    this.color = "Red";
  }

//Regular function:
  changeColor = function() {
    document.getElementById("demo").innerHTML += this;
  }
}

const myheader = new Header();

//The window object calls the function:
window.addEventListener("load", myheader.changeColor);

//A button object calls the function:
document.getElementById("btn").addEventListener("click", myheader.changeColor);
```

Example

With an arrow function, `this` represents the Header object no matter who called the function:

```
class Header {
  constructor() {
    this.color = "Red";
  }

//Arrow function:
  changeColor = () => {
    document.getElementById("demo").innerHTML += this;
  }
}

const myheader = new Header();


//The window object calls the function:
window.addEventListener("load", myheader.changeColor);

//A button object calls the function:
document.getElementById("btn").addEventListener("click", myheader.changeColor);
```

Remember these differences when you are working with functions. Sometimes the behavior of regular functions is what you want, if not, use arrow functions.

React ES6 Variables

Variables

Before ES6 there were only one way of defining your variables: with the `var` keyword. If you did not define them, they would be assigned to the global object. Unless you were in strict mode, then you would get an error if your variables were undefined.
Now, with ES6, there are three ways of defining your variables: `var`, `let`, and `const`.

var

```
var x = 5.6;
```

If you use `var` outside of a function, it belongs to the global scope.
If you use `var` inside of a function, it belongs to that function.
If you use `var` inside of a block, i.e. a for loop, the variable is still available outside of that block.
`var` has a *function* scope, not a *block* scope.

let

```
let x = 5.6;
```

`let` is the block scoped version of `var`, and is limited to the block (or expression) where it is defined.
If you use `let` inside of a block, i.e. a for loop, the variable is only available inside of that loop.
`let` has a *block* scope.

const

```
const x = 5.6;
```

`const` is a variable that once it has been created, its value can never change.
`const` has a *block* scope.
The keyword `const` is a bit misleading.
It does not define a constant value. It defines a constant reference to a value.
Because of this you can NOT:

- Reassign a constant value
- Reassign a constant array
- Reassign a constant object

But you CAN:

- Change the elements of constant array
- Change the properties of constant object

React ES6 Array Methods

Array Methods

There are many JavaScript array methods.
One of the most useful in React is the `.map()` array method.
The `.map()` method allows you to run a function on each item in the array, returning a new array as the result.
In React, `map()` can be used to generate lists.

Example

Generate a list of items from an array: index.js:

```
import React from 'react';
import ReactDOM from 'react-dom';
const myArray = ['apple', 'banana', 'orange'];
const myList = myArray.map((item) => <p>{item}</p>)
ReactDOM.render(myList, document.getElementById('root'));
```

React ES6 Destructuring

Destructuring

To illustrate destructuring, we'll make a sandwich. Do you take everything out of the refrigerator to make your sandwich? No, you only take out the items you would like to use on your sandwich.
Destructuring is exactly the same. We may have an array or object that we are working with, but we only need some of the items contained in these.
Destructuring makes it easy to extract only what is needed.

Destructing Arrays

Here is the old way of assigning array items to a variable:

Before:

```
const vehicles = ['mustang', 'f-150', 'expedition'];

// old way
const car = vehicles[0];
const truck = vehicles[1];
const suv = vehicles[2];
```

Here is the new way of assigning array items to a variable:

With destructuring:

```
const vehicles = ['mustang', 'f-150', 'expedition'];
const [car, truck, suv] = vehicles;
```

When destructuring arrays, the order that variables are declared is important.
If we only want the car and suv we can simply leave out the truck but keep the comma:

```
const vehicles = ['mustang', 'f-150', 'expedition'];
const [car,, suv] = vehicles;
```

Destructuring comes in handy when a function returns an array:

Example

```
function calculate(a, b) {
  const add = a + b;
  const subtract = a - b;
  const multiply = a * b;
  const divide = a / b;

  return [add, subtract, multiply, divide];
}
const [add, subtract, multiply, divide] = calculate(4, 7);
```

Destructuring Objects

Here is the old way of using an object inside a function:

Before:

```
const vehicleOne = {
  brand: 'Ford',
  model: 'Mustang',
  type: 'car',
  year: 2021,
  color: 'red'
}
myVehicle(vehicleOne);
// old way
function myVehicle(vehicle) {
  const message = 'My ' + vehicle.type + ' is a ' + vehicle.color + ' ' +
vehicle.brand + ' ' + vehicle.model + '.';
}
```

Here is the new way of using an object inside a function:

With destructuring:

```
const vehicleOne = {
  brand: 'Ford',
  model: 'Mustang',
  type: 'car',
  year: 2021,
  color: 'red'
}

myVehicle(vehicleOne);

function myVehicle({type, color, brand, model}) {
  const message = 'My ' + type + ' is a ' + color + ' ' + brand + ' ' + model + '.';
}
```

Notice that the object properties do not have to be declared in a specific order.

We can even destructure deeply nested objects by referencing the nested object then using a colon and curly braces to again destructure the items needed from the nested object:

Example

```
const vehicleOne = {
  brand: 'Ford',
  model: 'Mustang',
  type: 'car',
  year: 2021,
  color: 'red',
  registration: {
    city: 'Houston',
    state: 'Texas',
    country: 'USA'
  }
}
myVehicle(vehicleOne)
function myVehicle({ model, registration: { state } }) {
  const message = 'My ' + model + ' is registered in ' + state + '.';
}
```

React ES6 Spread Operator

Spread Operator

The JavaScript spread operator (`...`) allows us to quickly copy all or part of an existing array or object into another array or object.

Example

```
const numbersOne = [1, 2, 3];
const numbersTwo = [4, 5, 6];
const numbersCombined = [...numbersOne, ...numbersTwo];
```

The spread operator is often used in combination with destructuring.

Example

Assign the first and second items from `numbers` to variables and put the rest in an array:

```
const numbers = [1, 2, 3, 4, 5, 6];

const [one, two, ...rest] = numbers;
```

We can use the spread operator with objects too:

Example

Combine these two objects:

```
const myVehicle = {
  brand: 'Ford',
  model: 'Mustang',
  color: 'red'
}

const updateMyVehicle = {
  type: 'car',
  year: 2021,
  color: 'yellow'
}

const myUpdatedVehicle = {...myVehicle, ...updateMyVehicle}
```

Console log output:

```
Object
        brand: "Ford"
        color: "yellow"
        model: "Mustang"
        type: "car"
        year: 2021
        __proto__:
```

Notice the properties that did not match were combined, but the property that did match, `color`, was overwritten by the last object that was passed, `updateMyVehicle`. The resulting color is now yellow.

React ES6 Modules

Modules

JavaScript modules allow you to break up your code into separate files.
This makes it easier to maintain the code-base.
ES Modules rely on the `import` and `export` statements.

Export

You can export a function or variable from any file.
Let us create a file named `person.js`, and fill it with the things we want to export.
There are two types of exports: Named and Default.

Named Exports

You can create named exports two ways. In-line individually, or all at once at the bottom.

In-line individually:

person.js

```
export const name = "Jesse"
export const age = 40
```

All at once at the bottom:

person.js

```
const name = "Jesse"
const age = 40

export { name, age }
```

Default Exports

Let us create another file, named `message.js`, and use it for demonstrating default export.
You can only have one default export in a file.

Example

message.js

```
const message = () => {
  const name = "Jesse";
  const age = 40;
  return name + ' is ' + age + 'years old.';
};

export default message;
```

Import

You can import modules into a file in two ways, based on if they are named exports or default exports.
Named exports must be destructured using curly braces. Default exports do not.

Import from named exports

Import named exports from the file person.js:

```
import { name, age } from "./person.js";
```

Import from default exports

Import a default export from the file message.js:

```
import message from "./message.js";
```