Modern JavaScript: Recap for React

FbW IV

leandro.frigerio@devugees.org

21.2.2017

Modern JavaScript

In the old days, you could just include a <*script*> tag in the header of your webpage, and your JavaScript would run as intended. These days, we preprocess our JavaScript with Babel in order to access experimental features and language extensions like JSX.

ES5

ECMAScript is the language specification used to implement the JavaScript language. Nearly every JavaScript environment today can run at least ECMAScript 5 (ES5), the version of JavaScript introduced in **2009**. However, there are many new features in the latest versions of JavaScript that we'd like to use. Thanks to Babel, we can use them today! Babel transforms newer features into ES5 for cross-platform compatibility.

ES6

ES2015, or ECMAScript 2015, is the first significant update to the language since ES5 was initially released in 2009. You'll often see ES2015 called by its original name, ES6, since it's the 6th version of ECMAScript. Many ES2015 features are already available in modern JavaScript engines. However, for maximum browser compatibility, it's still safest to use Babel and compile down to ES5.

ES7 and further

There have already been many other features proposed for future versions of Javascript, including ES2016 (ES7) and ES2017 (ES8).

With Babel, we can use many of these features today.

Block Scoped Declarations

Instead of using var to declare local variables, we use const and let. The main difference is that var is scoped to a function, while const and let are scoped to a block.

Additionally, variables declared with const can only be assigned a value once. Assigning another value to the same name will throw a compiler error. Note that if the value assigned to a const variable is an object or array, the object or array may still be modified. In other words, it's only the variable name that is bound permanently -- the value itself is still mutable.

Using const and let

```
const a = 1
let b = 'foo'

// Not allowed!
// a = 2

// Ok!
b = 'bar'

if (true) {
  const a = 3
}
```

No Errors

Show Details

Output compiled with Babel

```
var a = 1;
var b = 'foo';

// Not allowed!
// a = 2

// Ok!
b = 'bar';

if (true) {
  var _a = 3;
}
```

Fat Arrow Functions

The fat arrow => is used to define anonymous functions. There are **two important differences** in the behavior of these functions, compared to functions defined with function.

First, the binding for the keyword this is the same outside and inside the fat arrow function. This is different than functions declared with function, which can bind this to another object upon invocation.

Maintaining this binding is very convenient for operations like mapping:
this.items.map(x => this.doSomethingWith(x)).

Second, fat arrow functions don't have an arguments object defined.
You can achieve the same thing using the spread syntax:

(...args) => doSomething(args[0], args[1]).

The fat arrow function syntax can vary a bit.

If the function takes exactly one parameter, the parentheses can be omitted:

x => Math.pow(x, 2).

Any other number of arguments will need parentheses:

(x, y) => Math.pow(x, y).

Fat Arrow Functions

If the function body is not wrapped in curly braces (as in the previous sentences), it is executed as an expression, and the return value of the function is the value of the expression. The function body can be wrapped in curly braces to make it a block, in which case you will need to explicitly return a value, if you want something returned. You will likely use the curly braces and block version more frequently, as this allows the function body to include multiple lines of code.

```
1  const foo = () => 'bar'
2
3  const baz = (x) => {
4    return x + 1
5  }
6
7  const squareSum = (...args) => {
8    const squared = args.map(x => Math.pow(x, 2))
9    return squared.reduce((prev, curr) => prev + curr)
10 }
11
12 this.items.map(x => this.doSomethingWith(x))
```

No Errors Show Details

```
1  var _this = this;
2
3  var foo = function foo() {
4    return 'bar';
5  };
6
7  var baz = function baz(x) {
8    return x + 1;
9  };
10
11 var squareSum = function squareSum() {
12   for (var _len = arguments.length, args = Array(_len), _key = 0; _key < _
13     args[_key] = arguments[_key];
14   }
```

Destructuring

Destructuring is a convenient way to extract multiple keys from an object or array simultaneously and assign the values to local variables.

Destructuring

```
1 const arr = ['one!', 'two!', 'three!', 'four!']
2 const [one, two, ...rest] = arr
3
4 const obj = {a: 'x', b: 'y', c: 'z'}
5 const {a, b, c} = obj
```

No Errors

Show Details

Output compiled with Babel

```
1 var arr = ['one!', 'two!', 'three!', 'four!'];
2 var one = arr[0],
3     two = arr[1],
4     rest = arr.slice(2);
5
6
7 var obj = { a: 'x', b: 'y', c: 'z' };
8 var a = obj.a,
9     b = obj.b,
10    c = obj.c;
```

Imports and Exports

ES2015 provides a more advanced module importing/exporting pattern than the widely used CommonJS pattern. By contrast to the old module.exports = {...}, we can now export multiple named values. Similarly, we can import multiple named values. There is one default export per file, and this exported value can be imported without refering to it by name. Every other import and export must be named.

Importing

```
// import the default export
import React from 'react'

// import other named exports
import { Component, Children } from 'react'

// import default and others simultaneously
// import React, { Component, Children } from 'react'
```

No Errors

Show Details

Output compiled with Babel

```
var _react = require('react');

var _react2 = _interopRequireDefault(_react);

function _interopRequireDefault(obj) { return obj && obj.__esModule ? obj :
```

Imports and Exports

The imports in the previous example would be available if exported from the module react as in the next example.

Exporting

```
1 export default React
2 export { Component, Children }
```

No Errors

Show Details

Output compiled with Babel

```
1 Object.defineProperty(exports, "__esModule", {
2   value: true
3 });
4 exports.default = React;
5 exports.Component = Component;
6 exports.Children = Children;
```

For full details on the importing/exporting syntax, see the MDN reference for import and export.

Default Parameters

We can assign default values to function parameters within the function declaration. A default value is assigned to a parameter if it is **undefined**.

Default parameters

```
1 const printInput = (input = 'hello world') => {
2   console.log(input)
3 }
4
5 printInput()
6 printInput('hello universe')
```

No Errors

Show Details

Console output

```
"hello world"
"hello universe"
```

Classes

In ES5, classes are written as functions, with instance methods assigned to **MyFunction.prototype**. ES2015 allows us to use the simpler **class** syntax.

class gives us built in instance functions, static functions, and inheritance. **constructor** is a special function that is called automatically every time a class instance is created.

We can use the **static** keyword to declare static class functions. Static method calls are made directly on the class and cannot be called on instances of the class.

Using class

```
class Calculator {
  constructor(value1, value2) {
    this.value1 = value1
    this.value2 = value2
  }

  static multiply(value1, value2) {
    return value1 * value2
  }

  sum() {
    return this.value1 + this.value2
  }
```

No Errors

Show Details

Console output

```
5
6
```

Inheritance

Class gives us simple inheritance with the keyword **extends**. Classes that inherit from a parent have access to respective parent functions via **super**.

Inheritance

```
class SquareCalculator {
  constructor(value) {
    this.value = value
  }

  calculate() {
    return this.value * this.value
  }
}

class CubeCalculator extends SquareCalculator {
  calculate() {
```

No Errors

Show Details

Console output

```
27
```

For full details on the class syntax, see the MDN reference.

Dynamic Object Keys

In ES5, object literal keys are always interpreted as a string.

ES2015 allows us to use computed values as keys in object literals, using square bracket syntax: [**myKey**].

Dynamic object keys

```
1 const attribute = 'color'
2 const style = {
3   [attribute]: 'white'
4 }
5
6 console.log(style.color)
```

No Errors

Show Details

Console output

```
"white"
```

Array Spread

The array spread syntax **...** makes it easy to expand an array.

This can be used to make a shallow copy of an array. It also provides a succinct way to concatenate and unshift arrays.

Array spread

```
1 const foo = ['a', 'b', 'c']
2 const bar = ['d', 'e', 'f']
3
4 console.log(...foo)
5 console.log([...foo, ...bar])
6
```

No Errors

Show Details

Console output

```
"a" "b" "c"
▶ ["a", "b", "c", "d", "e", "f"]
```

ES6 - Static Class Properties

As we saw in our ES2015 section, static functions on classes exist as a part of ES2015. In ES2016, we can use the static keyword to declare static properties as well. Static properties exist directly on the class.

Static properties

```
1 class Foo {
2   static bar = 'hello'
3 }
4
5 console.log(Foo.bar)
```

No Errors

Show Details

Console output

```
"hello"
```

ES6 - Class Instance Properties

Class instance properties are a convenient way to declare properties for each instance, equivalent to assigning these properties in the constructor function.

Class instance properties

```
class Cat {
  name = 'Tom'
  state = {
    running: true
  }

  constructor() {
    console.log(this.name, this.state.running)
  }
}

new Cat()
```

No Errors

Show Details

Console output

```
"Tom" true
```

Note that this language feature is currently in the proposal stage (not officially adopted as part of the language yet). It's a "stage 2" proposal, meaning it's unlikely to change. Read more about this here.

ES6 - Bound Instance Methods

When a function is assigned to a class instance property, that function is bound to the instance.

Before ES2016, you might bind functions to class instances in the **constructor**,

this.func = this.func.bind(this)

Binding here ensures that a class's instance function is invoked with the correct context.

With ES2016 class instance properties, we can instead write **func = () =>**. func is then bound to the class instance at construction.

Bound instance methods

```
class Cat {
  constructor(name) {
    this.name = name
  }
  printName = () => {    console.log(this.name) }
}
const cat = new Cat('Tom')
const printName = cat.printName
// 'this' is still bound to our Cat instance, so even
// though our calling context changed, the function
// executes in its original context.
printName()
```

No Errors

Show Details

Console output

```
"Tom"
```

ES6 - Object Spread

Similar to the array spread operator in ES2015, ES2016 offers a spread operator ... for objects. This tries to use ES2015's Object.assign, as you'll see when you view the babel output of the spread operator. This can be very useful in copying or extending objects.

We can copy an object simply with {...originalObj}. Note that this is a shallow copy. We can also extend an object with {...originalObj, key1: 'newValue'}. Similarly to assign, when duplicate keys appear in a spread, the last assignment of that key takes priority.

Object spread operator

```
const defaultStyle = {
  color: 'black',
  fontSize: 12,
  fontWeight: 'normal'
}
const style = {
  ...defaultStyle,
  fontWeight: 'bold',
  backgroundColor: 'white'
}

// Note that fontWeight is 'bold', as the second assignment overrode the initial assignment.
console.log(style)
```

No Errors

Show Details

Console output

```
▸Object {color: "black",
fontSize: 12, fontWeight: "bold",
backgroundColor: "white"}
```

ES6 - Async and Await

We can use the async keyword before a function name to wrap the return value of this function in a Promise. We can use the await keyword (in an async function) to wait for a promise to be resolved or rejected before continuing code execution in this block.

This syntax also propagates exceptions that occur in promises using a try/catch block, just as if the code were running synchronously.

Async and await

```
1 const fetchData = async () => {
2   return fetch('https://randomuser.me/api/')
3 }
4 const printData = async () => {
5   try {
6     const data = await fetchData()
7     const json = await data.json()
8     console.log(json)
9   } catch(e) {
10    console.error("Problem", e)
11  }
12 }
13 printData()
```

No Errors

Show Details

Console output

```
▸Object {results: Array[1],
info: Object}
```

EJX

JSX is an extension to JavaScript that adds a new kind of expression.

You can use JSX expressions anywhere you could use any other expression.

JSX is a shortcut for using the React.createElement() API, that is much more concise, easy to read, and visually looks a little like the generated UI (as both are tree-like). You don't have to use JSX, but there are practically no disadvantages, so you probably should use it.

JSX is tag-based like XML or HTML.

Each tag, like , is transformed into a call to React.createElement().

Any attributes become props of the instantiated component. Attributes can be strings like foo='hello', or they can be interpolated JavaScript expressions when wrapped in curly braces as in bar={baz} (which would set the bar prop to the variable baz).

Tags can be self-closing, like , or they can include both an opening and closing tag, like . To include children elements, you will need to use an opening and closing tag and put the children tags within.

JSX

JSX

```
1 const a = <div />
2
3 const b = (
4   <div
5     foo='hello'
6     bar={baz}>
7     <span>42</span>
8   </div>
9 )
```

No Errors

Show Details

Output compiled with Babel

```
1 var a = React.createElement('div', null);
2
3 var b = React.createElement(
4   'div',
5   {
6     foo: 'hello',
7     bar: baz },
8   React.createElement(
9     'span',
10    null,
11    '42'
12  )
13 );
```

Resources

- http://www.react.express
  Devin Abbott @dvnabbott
- https://babeljs.io/learn-es2015/
  ECMA Scripts 2015
- https://github.com/lukehoban/es6features
  ES6 Features
- http://exploringjs.com/
  Exploring ES6
- https://www.youtube.com/watch?v=hO7mzO83N1Q&feature=youtu.be&t=39m35s
  JavaScript Patterns for 2017 - Scott Allen
- https://www.w3resource.com/slides/ecmascript-6-style-guide.php
  ES6 Style Guide