I recently went to [HTML5 Dev conference](http://webapplog.com/html5) in San Francisco. Half of the talks I went to were about ES6 or, as it’s now called officially, ECMAScript2015. I prefer the more succinct ES6 though.

This essay will give you a quick introduction to ES6. If you don’t know what is ES6, it’s a new JavaScript implementation. If you’re a busy JavaScript software engineer (and who is not?), then proceed reading to learn the best 10 features of the new generation of the most popular programming language—JavaScript.

Here’s the list of the top 10 best ES6 features for a busy software engineer (in no particular order):

1. Default Parameters in ES6
2. Template Literals in ES6
3. Multi-line Strings in ES6
4. Destructuring Assignment in ES6
5. Enhanced Object Literals in ES6
6. Arrow Functions in ES6
7. Promises in ES6
8. Block-Scoped Constructs Let and Const
9. Classes in ES6
10. Modules in ES6

Disclaimer: the list if highly biased and subjective. It is in no way was intended to diminish usefulness of other ES6 features, which didn’t make it to the list simply because I had to limit the number to 10.

First, a bit of history because those who don’t know the history can’t make it. This is a brief JavaScript timeline:

1. 1995: JavaScript is born as LiveScript
2. 1997: ECMAScript standard is established
3. 1999: ES3 comes out and IE5 is all the rage
4. 2000–2005: XMLHttpRequest, a.k.a. AJAX, gains popularity in app such as Outlook Web Access (2000) and Oddpost (2002), Gmail (2004) and Google Maps (2005).
5. 2009: ES5 comes out (this is what most of us use now) with forEach,Object.keys, Object.create (specially for [Douglas Crockford](http://www.crockford.com/)), and standard JSON
6. 2015: ES6/ECMAScript2015 comes out; it has mostly syntactic sugar, because people weren’t able to agree on anything more ground breaking (ES7?)

Enough with history, let’s get to the business of coding.

**1. Default Parameters in ES6**

Remember we had to do these statements to define default parameters:

var link = function (height, color, url) {

var height = height || 50

var color = color || 'red'

var url = url || 'http://azat.co'

...

}

They were okay until the value was 0 and because 0 is falsy in JavaScript it would default to the hard-coded value instead of becoming the value itself. Of course, who needs 0 as a value (#sarcasmfont), so we just ignored this flaw and used the logic OR anyway… No more! In ES6, we can put the default values right in the signature of the functions:

var link = function(height = 50, color = 'red', url = 'http://azat.co') {

...

}

By the way, this syntax is similar to Ruby!

**2. Template Literals in ES6**

Template literals or interpolation in other languages is a way to output variables in the string. So in ES5 we had to break the string like this:

var name = 'Your name is ' + first + ' ' + last + '.'

var url = 'http://localhost:3000/api/messages/' + id

Luckily, in ES6 we can use a new syntax ${NAME} inside of the back-ticked string:

var name = `Your name is ${first} ${last}.`

var url = `http://localhost:3000/api/messages/${id}`

**3. Multi-line Strings in ES6**

Another yummy syntactic sugar is multi-line string. In ES5, we had to use one of these approaches:

var roadPoem = 'Then took the other, as just as fair,\n\t'

+ 'And having perhaps the better claim\n\t'

+ 'Because it was grassy and wanted wear,\n\t'

+ 'Though as for that the passing there\n\t'

+ 'Had worn them really about the same,\n\t'

var fourAgreements = 'You have the right to be you.\n\

You can only be you when you do your best.'

While in ES6, simply utilize the backticks:

var roadPoem = `Then took the other, as just as fair,

And having perhaps the better claim

Because it was grassy and wanted wear,

Though as for that the passing there

Had worn them really about the same,`

var fourAgreements = `You have the right to be you.

You can only be you when you do your best.`

**4. Destructuring Assignment in ES6**

Destructuring can be a harder concept to grasp, because there’s some magic going on… let’s say you have simple assignments where keys house and mouseare variables house and mouse:

var data = $('body').data(), // data has properties house and mouse

house = data.house,

mouse = data.mouse

Other examples of destructuring assignments (from Node.js):

var jsonMiddleware = require('body-parser').json

var body = req.body, // body has username and password

username = body.username,

password = body.password

In ES6, we can replace the ES5 code above with these statements:

var { house, mouse} = $('body').data() // we'll get house and mouse variables

var {jsonMiddleware} = require('body-parser')

var {username, password} = req.body

This also works with arrays. Crazy!

var [col1, col2] = $('.column'),

[line1, line2, line3, , line5] = file.split('\n')

It might take some time to get use to the destructuring assignment syntax, but it’s a sweet sugarcoating.

**5. Enhanced Object Literals in ES6**

What you can do with object literals now is mind blowing! We went from a glorified version of JSON in ES5 to something closely resembling classes in ES6.

Here’s a typical ES5 object literal with some methods and attributes/properties:

var serviceBase = {port: 3000, url: 'azat.co'},

getAccounts = function(){return [1,2,3]}

var accountServiceES5 = {

port: serviceBase.port,

url: serviceBase.url,

getAccounts: getAccounts,

toString: function() {

return JSON.stringify(this.valueOf())

},

getUrl: function() {return "http://" + this.url + ':' + this.port},

valueOf\_1\_2\_3: getAccounts()

}

If we want to be fancy, we can inherit from serviceBase by making it the prototype with the Object.create method:

var accountServiceES5ObjectCreate = Object.create(serviceBase)

var accountServiceES5ObjectCreate = {

getAccounts: getAccounts,

toString: function() {

return JSON.stringify(this.valueOf())

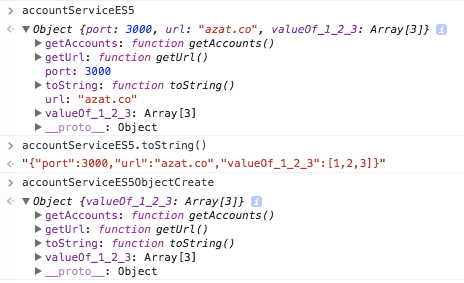
},

getUrl: function() {return "http://" + this.url + ':' + this.port},

valueOf\_1\_2\_3: getAccounts()

}

I know, accountServiceES5ObjectCreate and accountServiceES5 are NOT totally identical, because one object (accountServiceES5) will have the properties in the\_\_proto\_\_ object as shown below:

[](http://m03s6dh33i0jtc3uzfml36au.wpengine.netdna-cdn.com/wp-content/uploads/s-pYuaRxwA0HKoW-BmPPBmWARd9qzHY-dqIRyWg1mhU.png)

*Enhanced Object Literals in ES6*

But for the sake of the example, we’ll consider them similar. So in ES6 object literal, there are shorthands for assignment getAccounts: getAccounts,becomes just getAccounts,. Also, we set the prototype right there in the\_\_proto\_\_`` property which makes sense (not‘**proto**’` though:

var serviceBase = {port: 3000, url: 'azat.co'},

getAccounts = function(){return [1,2,3]}

var accountService = {

\_\_proto\_\_: serviceBase,

getAccounts,

Also, we can invoke super and have dynamic keys (valueOf\_1\_2\_3):

toString() {

return JSON.stringify((super.valueOf()))

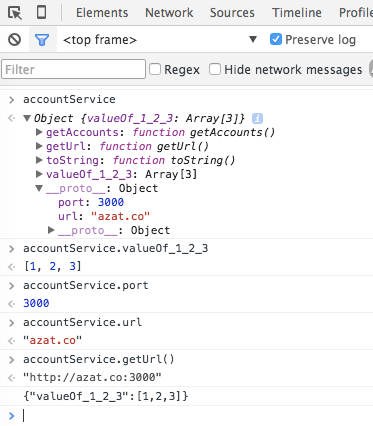
},

getUrl() {return "http://" + this.url + ':' + this.port},

[ 'valueOf\_' + getAccounts().join('\_') ]: getAccounts()

};

console.log(accountService)

[](http://m03s6dh33i0jtc3uzfml36au.wpengine.netdna-cdn.com/wp-content/uploads/woRH94fOhCC0aYxgI40qu-dUzHPoxNjhwsLG5U4ob0c.png)

*Enhanced Object Literals in ES6 II*

This is a great enhancement to good old object literals!

**6. Arrow Functions in ES6**

This is probably one feature I waited the most. I love CoffeeScript for its fat arrows. Now we have them in ES6. The fat arrows are amazing because they would make your this behave properly, i.e., this will have the same value as in the context of the function—it won’t mutate. The mutation typically happens each time you create a closure.

Using arrows functions in ES6 allows us to stop using that = this or self = this or \_this = this or .bind(this). For example, this code in ES5 is ugly:

var \_this = this

$('.btn').click(function(event){

\_this.sendData()

})

This is the ES6 code without \_this = this:

$('.btn').click((event) =>{

this.sendData()

})

Sadly, the ES6 committee decided that having skinny arrows is too much of a good thing for us and they left us with a verbose old function instead. ([Skinny arrow in CoffeeScript](https://www.udemy.com/coffeescript/?couponCode=a) works like regular function in ES5 and ES6).

Here’s another example in which we use call to pass the context to thelogUpperCase() function in ES5:

var logUpperCase = function() {

var \_this = this

this.string = this.string.toUpperCase()

return function () {

return console.log(\_this.string)

}

}

logUpperCase.call({ string: 'es6 rocks' })()

While in ES6, we don’t need to mess around with \_this:

var logUpperCase = function() {

this.string = this.string.toUpperCase()

return () => console.log(this.string)

}

logUpperCase.call({ string: 'es6 rocks' })()

Note that you can mix and match old function with => in ES6 as you see fit. And when an arrow function is used with one line statement, it becomes an expression, i.e,. it will implicitly return the result of that single statement. If you have more than one line, then you’ll need to use return explicitly.

This ES5 code is creating an array from the messages array:

var ids = ['5632953c4e345e145fdf2df8','563295464e345e145fdf2df9']

var messages = ids.map(function (value) {

return "ID is " + value // explicit return

});

Will become this in ES6:

var ids = ['5632953c4e345e145fdf2df8','563295464e345e145fdf2df9']

var messages = ids.map(value => `ID is ${value}`) // implicit return

Notice that I used the string templates? Another feature from CoffeeScript… I love them!

The parenthesis () are optional for single params in an arrow function signature. You need them when you use more than one param.

In ES5 the code has function with explicit return:

var ids = ['5632953c4e345e145fdf2df8', '563295464e345e145fdf2df9'];

var messages = ids.map(function (value, index, list) {

return 'ID of ' + index + ' element is ' + value + ' ' // explicit return

});

And more eloquent version of the code in ES6 with parenthesis around params and implicit return:

var ids = ['5632953c4e345e145fdf2df8','563295464e345e145fdf2df9']

var messages = ids.map((value, index, list) => `ID of ${index} element is ${value} `) // implicit return

**7. Promises in ES6**

Promises have been a controversial topic. There were a lot of promise implementations with slightly different syntax. q, bluebird, deferred.js, vow, avow, jquery deferred to name just a few. Others said we don’t need promises and can just use async, generators, callbacks, etc. Gladly, there’s a standardPromise implementation in ES6 now!

Let’s consider a rather trivial example of a delayed asynchronous execution with setTimeout():

setTimeout(function(){

console.log('Yay!')

}, 1000)

We can re-write the code in ES6 with Promise:

var wait1000 = new Promise(function(resolve, reject) {

setTimeout(resolve, 1000)

}).then(function() {

console.log('Yay!')

})

Or with ES6 arrow functions:

var wait1000 = new Promise((resolve, reject)=> {

setTimeout(resolve, 1000)

}).then(()=> {

console.log('Yay!')

})

So far, we’ve increased the number of lines of code from three to five without any obvious benefit. That’s right. The benefit will come if we have more nested logic inside of the setTimeout() callback:

setTimeout(function(){

console.log('Yay!')

setTimeout(function(){

console.log('Wheeyee!')

}, 1000)

}, 1000)

Can be re-written with ES6 promises:

var wait1000 = ()=> new Promise((resolve, reject)=> {setTimeout(resolve, 1000)})

wait1000()

.then(function() {

console.log('Yay!')

return wait1000()

})

.then(function() {

console.log('Wheeyee!')

});

Still not convinced that Promises are better than regular callbacks? Me neither. I think once you got the idea of callbacks and wrap your head around them, then there’s no need for additional complexity of promises.

Nevertheless, ES6 has Promises for those of you who adore them. Promises have a fail-and-catch-all callback as well which is a nice feature. Take a look at this post for more info on promises: [*Introduction to ES6 Promises*](http://jamesknelson.com/grokking-es6-promises-the-four-functions-you-need-to-avoid-callback-hell).

**8. Block-Scoped Constructs Let and Const**

You might have already seen the weird sounding let in ES6 code. I remember the first time I was in London, I was confused by all those TO LET signs. The ES6 let has nothing to do with renting. This is not a sugarcoating feature. It’s more intricate. let is a new var which allows to scope the variable to the blocks. We define blocks by the curly braces. In ES5, the blocks did NOTHING to the vars:

function calculateTotalAmount (vip) {

var amount = 0

if (vip) {

var amount = 1

}

{ // more crazy blocks!

var amount = 100

{

var amount = 1000

}

}

return amount

}

console.log(calculateTotalAmount(true))

The result will be 1000. Wow! That’s a really bad bug. In ES6, we use let to restrict the scope to the blocks. Vars are function scoped.

function calculateTotalAmount (vip) {

var amount = 0 // probably should also be let, but you can mix var and let

if (vip) {

let amount = 1 // first amount is still 0

}

{ // more crazy blocks!

let amount = 100 // first amount is still 0

{

let amount = 1000 // first amount is still 0

}

}

return amount

}

console.log(calculateTotalAmount(true))

The value is 0, because the if block also has let. If it had nothing (amount=1), then the expression would have been 1.

When it comes to const, things are easier; it’s just an immutable, and it’s also block-scoped like let. Just to demonstrate, here are a bunch of constants and they all are okay because they belong to different blocks:

function calculateTotalAmount (vip) {

const amount = 0

if (vip) {

const amount = 1

}

{ // more crazy blocks!

const amount = 100

{

const amount = 1000

}

}

return amount

}

console.log(calculateTotalAmount(true))

In my humble opinion, let and const overcomplicate the language. Without them we had only one behavior, now there are multiple scenarios to consider. ;-(

**9. Classes in ES6**

If you love object-oriented programming (OOP), then you’ll love this feature. It makes writing classes and inheriting from them as easy as liking a comment on Facebook.

Classes creation and usage in ES5 was a pain in the rear, because there wasn’t a keyword class (it was reserved but did nothing). In addition to that, lots of inheritance patterns like [pseudo classical](http://javascript.info/tutorial/pseudo-classical-pattern), [classical](http://www.crockford.com/javascript/inheritance.html), [functional](http://javascript.info/tutorial/factory-constructor-pattern) just added to the confusion, pouring gasoline on the fire of religious JavaScript wars.

I won’t show you how to write a class (yes, yes, there are classes, objects inherit from objects) in ES5, because there are many flavors. Let’s take a look at the ES6 example right away. I can tell you that the ES6 class will use prototypes, not the function factory approach. We have a class baseModel in which we can define a constructor and a getName() method:

class baseModel {

constructor(options = {}, data = []) { // class constructor

this.name = 'Base'

this.url = 'http://azat.co/api'

this.data = data

this.options = options

}

getName() { // class method

console.log(`Class name: ${this.name}`)

}

}

Notice that I’m using default parameter values for options and data. Also, method names don’t need to have the word function or the colon (:) anymore. The other big difference is that you can’t assign properties this.NAME the same way as methods, i.e., you can’t say name at the same indentation level as a method. To set the value of a property, simply assign a value in the constructor.

The AccountModel inherits from baseModel with class NAME extends PARENT\_NAME:

class AccountModel extends baseModel {

constructor(options, data) {

To call the parent constructor, effortlessly invoke super() with params:

super({private: true}, ['32113123123', '524214691']) //call the parent method with super

this.name = 'Account Model'

this.url +='/accounts/'

}

If you want to be really fancy, you can set up a getter like this and accountsDatawill be a property:

get accountsData() { //calculated attribute getter

// ... make XHR

return this.data

}

}

So how do you actually use this abracadabra? It’s as easy as tricking a three-year old into thinking Santa Claus is real:

let accounts = new AccountModel(5)

accounts.getName()

console.log('Data is %s', accounts.accountsData)

In case you’re wondering, the output is:

Class name: Account Model

Data is %s 32113123123,524214691

**10. Modules in ES6**

As you might now, there were no native modules support in JavaScript before ES6. People came up with AMD, RequireJS, CommonJS and other workarounds. Now there are modules with import and export operands.

In ES5 you would use <script> tags with IIFE, or some library like AMD, while in ES6 you can expose your class with export. I am a Node.js guy, so I’ll use CommonJS which is also a Node.js syntax. It’s straightforward to use CommonJS on the browser with the [Browserify](http://browserify.org/) bunder. Let’s say we have portvariable and getAccounts method in ES5 module.js:

module.exports = {

port: 3000,

getAccounts: function() {

...

}

}

In ES5 main.js, we would require('module') that dependency:

var service = require('module.js')

console.log(service.port) // 3000

In ES6, we would use export and import. For example, this is our library in the ES6 module.js file:

export var port = 3000

export function getAccounts(url) {

...

}

In the importer ES6 file main.js, we use import {name} from 'my-module'syntax. For example,

import {port, getAccounts} from 'module'

console.log(port) // 3000

Or we can import everything as a variable service in main.js:

import \* as service from 'module'

console.log(service.port) // 3000

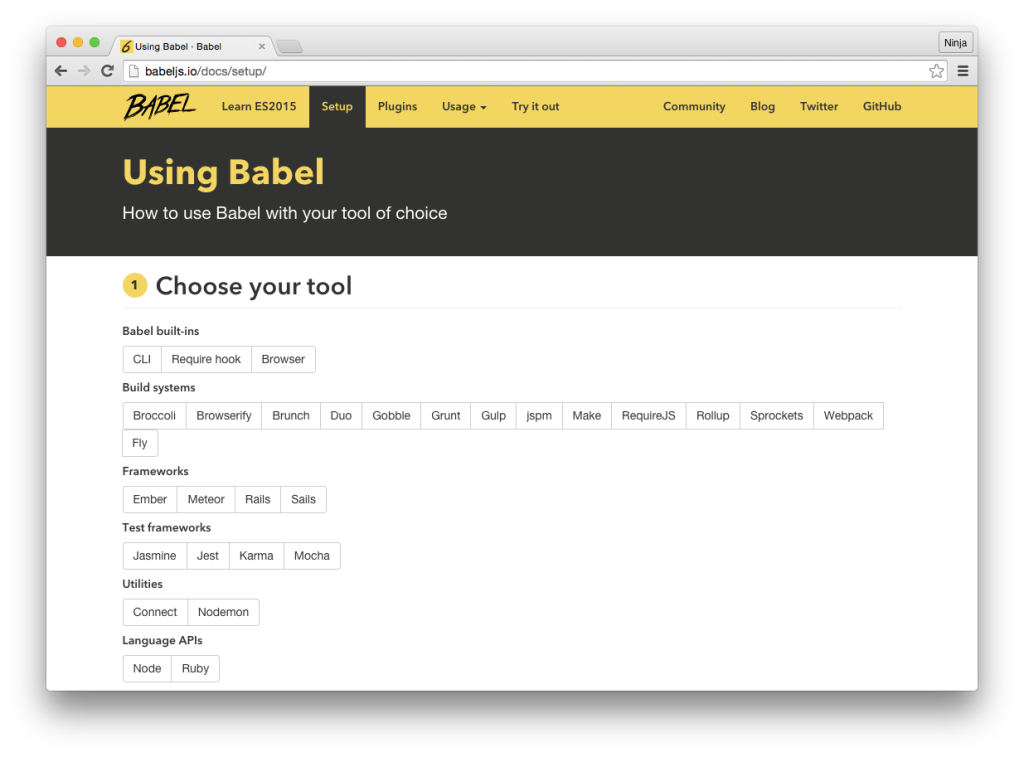
Personally, I find the ES6 modules confusing. Yes, they are more eloquent, but Node.js modules won’t change anytime soon. It’s better to have only one style for browser and server JavaScript, so I’ll stick with CommonJS/Node.js style for now.

The support for ES6 modules in the browsers are not coming anytime soon (as of this writing), so you’ll need something like [jspm](http://jspm.io/) to use ES6 modules.

For more information and examples on ES6 modules, take a look at [this text](http://exploringjs.com/es6/ch_modules.html). No matter what, write modular JavaScript!

**How to Use ES6 Today (Babel)**

ES6 is finalized, but not fully supported by all browsers (e.g., [ES6 Firefox support](https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_6_support_in_Mozilla)). To use ES6 today, get a compiler like [Babel](https://babeljs.io/). You can run it as a standalone tool or use with your build system. There are Babel [plugins](http://babeljs.io/docs/setup) for Grunt, Gulp and Webpack.

[](http://m03s6dh33i0jtc3uzfml36au.wpengine.netdna-cdn.com/wp-content/uploads/i2AGKyI1Jx5QZnOPrrF2SuyPY9zCCdUPitaBI-KMeyc.png)

*How to Use ES6 Today (Babel)*

Here’s a Gulp example. Install the plugin:

$ npm install --save-dev gulp-babel

In gulpfile.js, define a task build that takes src/app.js and compiles it into the build folder:

var gulp = require('gulp'),

babel = require('gulp-babel')

gulp.task('build', function () {

return gulp.src('src/app.js')

.pipe(babel())

.pipe(gulp.dest('build'))

})

**Node.js and ES6**

For Node.js, you can compile your Node.js files with a build tool or use a standalone Babel module babel-core. To install it,

$ npm install --save-dev babel-core

Then in Node.js, you call this function:

require("babel-core").transform(es5Code, options)

**Summary of ES6 Things**

There are many other noteworthy ES6 features which you probably won’t use (at least not right away). In no particular order:

1. New Math, Number, String, Array and Object methods
2. Binary and octal number types
3. Default rest spread
4. For of comprehensions (hello again mighty CoffeeScript!)
5. Symbols
6. Tail calls
7. Generators
8. New data structures like Map and Set