Top ES6/ES2015 Features

Section 2: Syntax Features

Azat Mardan @azat_co

© NodeProgram.com, Node.University and Azat Mardan 2017

Block-Scoped Constructs: Let and Const

© NodeProgram.com, Node.University and Azat Mardan 2017

`let` is a new `var` which allows developers to scope the variable to the logical blocks

We define blocks by the curly braces.

© NodeProgram.com, Node.University and Azat Mardan 2017

In ES5, the blocks did NOTHING to the vars as seen here:

```
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)) // 1000
```

© NodeProgram.com, Node.University and Azat Mardan 2017

In ES6, we use `let` to restrict the scope to the blocks. Variables are then 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)) // 0
```

When it comes to `const`, things are easier; it just prevent re-assigning, and it's also block-scoped like `let`.

In ES6, `const` prevents from re-assigning

```
const a = 1 //  ok
a = 2 // Uncaught TypeError: Assignment to constant variable.
```

`let` is ok with re-assigning:

```
let b = 1 // ok
b = 2 // b is 2
```

However, both `const` and `let` won't allow re-declaration:

```
const a = 1 // ok
// ...
const a = 1 // Uncaught SyntaxError: Identifier 'a' has already been declared
```

Both `const` and `let` won't allow re-declaration

```
let b = 1 // ok
// ...
let b = 1 // Uncaught SyntaxError: Identifier 'b' has already been declared
```

`let` and `const` are block-scoped but the latter prevent re-assigning

Rule of Thumb

- `const` for modules, classes, functions
- `let` for all mutable variable where you used to write `var`
- `var` is still okay, it won't break

`const` and `let` are about protection and predictability. 😄

Default Parameters

When we write classes/functions/modules, we need to account for cases when the arguments will be omitted.

In ES5, you would write something like the code below which uses the logical OR (||):

```
var getAccounts = function(limit) {
  var limit = limit || 10
  // ...
}

var link = function (height, color, url) {
    var height = height || 50
    var color = color || 'red'
    var url = url || 'https://node.university'
    // ...
}
```

Or maybe event an `if/else` condition to check for the undefined value:

```
var getAccounts = function(limit) {
  if (typeof limit == 'undefined') limit = 10
  // ...
  if (!color)
    color = 'red'
}
```

In ES6, we can put the default values right in the signature of the functions like so:

```
const getAccounts = function(limit = 10) {
  ...
}
const link = function(height = 50, color = 'red', url = 'https://node.university') {
  ...
}
```

Default parameters are about readability and predictability.

Rest and Spread Parameters

`arguments` object is not a real array. You have to convert it to an array if you wanted to use functions like `sort` or `map` explicitly:

```
var request = function request(url, options, callback) {
  var args = Array.prototype.slice.call(arguments)
  var url = args[0] // same as url
  var options = args[1] // same as options
  var callback = args[2]
  // ...
}
```

In ES6, use `...` in the function signature (rest parameter), e.g., `argumentsArray` become an array containing arguments/parameters starting after `options`, i.e., 3, 4, until N:

```
const request = function(url, options, ...argumentsArray) {
  let callback = argumentsArray[0]
  let extraArgument1 = argumentsArray[1]
  argumentsArray.map(function(argument){
    // ...
  })
  // ...
}
```

Rest vs. `arguments`: rest is for un-named argument while `arguments` will have both named and un-named arguments.

Putting other named arguments after the rest parameter will cause a **syntax error**.

The rest parameter is a great sugar-coating addition to JavaScript. It replaces the `arguments` object which wasn't a real array

© NodeProgram.com, Node.University and Azat Mardan 2017

Spread

In ES5, if you wanted to use an array as an argument to a function, you would have to use the `apply()` function:

```
function request(url, options, callback) {
  // ...
}
var requestArgs = ['https://node.university', {...}, function() {...}]
request.apply(null, requestArgs)
```

In ES6, we can use the spread parameters which look similar to the rest parameters in syntax as they use ellipses ...:

```
function request(url, options, callback) {
 // ...
}
const requestArgs = ['https://node.university', {...}, function() {...}]
request(...requestArgs) // <--- Function call with Array
```

ES6 developers can use the spread operator in the following cases

Function calls as seen above

Array literals, e.g., `array2 = [...array1, x, y, z]`

Destructuring (section 5 of this essay)

`new` function calls (constructors), e.g., `var d = new Date(...dates)`

`push()` calls, e.g., `arr1.push(...arr2)`

© NodeProgram.com, Node.University and Azat Mardan 2017

The spread operator has a similar syntax the rest parameters.

© NodeProgram.com, Node.University and Azat Mardan 2017

But rest is used in the function definition/declaration and spread is used in the calls and literals.

© NodeProgram.com, Node.University and Azat Mardan 2017

☝️ Rest in function declarations vs Spread in function calls & literals

"..." operators save developers from typing extra lines of imperative code, so knowing and using them is a good skill.

© NodeProgram.com, Node.University and Azat Mardan 2017

Template Literals

Template literals or interpolation as they're known in other languages are a way to output variables in the string mixed with some text; typically in user interfaces.

In ES5, we had to break the string:

```
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:

```
const first = 'Azat', // var is ok too
  last = 'Mardan',
  id = 777
let name = `Your name is ${first} ${last}.` // var is ok too
let url = `http://localhost:3000/api/messages/${id}`
```

Results:

» name is "Your name is Azat Mardan."

» url is "http://localhost:3000/api/messages/777"

© NodeProgram.com, Node.University and Azat Mardan 2017

How to write backtick as inline code in Markdown?

Text the code ``var roadPoem = `Then...` ``

© NodeProgram.com, Node.University and Azat Mardan 2017

Backticks are used in Markdown for inline code… how to work around the conflict?

To be able to write text and inline code with backticks inside

For example this: `` var roadPoem = `Then...` ``

Use two or three backticks for code and one inside for template literals:

```
For example this: ``var roadPoem = `Then...` ``
```

This is neat and allows developers to see the end result of the strings at one glance instead of trying to evaluate the concatenation expression.

© NodeProgram.com, Node.University and Azat Mardan 2017

Interpolation is good, but how do you work with multi-line text in JavaScript?

Multi-line Strings

© NodeProgram.com, Node.University and Azat Mardan 2017

In ES5, we had to use one of these approaches and it was ugly. With concatenation:

```
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'
```

Or with escape slash:

```
var fourAgreements = 'You have the right to be you.\n\
    You can only be you when you do your best.'
```

In ES6, simply utilize the backticks as follows:

```
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.`
```

Multi-line strings are a useful addition if you have to use a lot of text in your JavaScript code.

Destructuring Assignment in ES6+

Let's say you have simple assignments where keys `userId` and `accountNumber` are variables `userId` and `accountNumber`:

```
var data = $('body').data(), // data has properties userId and accountNumber
  userId = data.userId, // ok
  accountNumber = data.accountNumber // ok
```

Other examples of assignments where names of the variables and object properties are the same:

```
var json = require('body-parser').json // body-parser has json
//...
var body = req.body, // body has username and password
  username = body.username,
  password = body.password
```

As long as names to the left and to the right of the assignement are the same, i.e., variable name and the name of the property in the object, then we can apply *destructuring*.

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

```
var {userId, accountNumber} = $('body').data() // or const or let


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


var {username, password} = req.body
```

More examples from React-land (checkout the React Foundation course on NodeU)

```
const {render} = require('react-dom')
const {Router, Route, hashHistory} = require('react-router')
const {Navbar, NavItem, Nav} = require('react-bootstrap')
```

This also works with arrays. Crazy!

```
var [col1, col2]  = $('.column'),
  [line1, line2, line3, , line5] = file.split('\n')
```

It might take some time to get used to the destructuring assignment syntax, but it's a sweet sugarcoating nevertheless.

(Fat) Arrow Functions

This is probably the feature I waited on the most. I loved CoffeeScript for its fat arrows. Now we have them in ES6. ;-)

(Fat) Array ES6 Functions is a way to create an anonymous function.

```
var f = function(data) { // ES5
  // ...
  return data
}

const f = (data) => { // ES6+
  // ...
  return data
}
```

But there's more to (fat) arrow functions!

Do you understand how `this` works?

`this` can change value, especially when you create a new function.

© NodeProgram.com, Node.University and Azat Mardan 2017

jQuery will invoke the event handler and `this` will have a different value:

```
this.sendData = function() {...}
$('.btn').click(function(event){
  this.sendData() // Fail
})
```

Using arrows functions in ES6 allows us to stop using `that = this` or `self = this` or `_this = this` or `.bind(this)`.

In ES5, using `this` is kind of ugly because you can forget to transfer the context to the closure with `_this`:

```
var _this = this
$('.btn').click(function(event){
  _this.sendData()
})
```

The `bind()` or `call()` approaches are not much better because of their verbosity:

```
$('.btn').click(function(event){
  this.sendData()
}.bind(this))
```

But take a look at this pretty ES6 code:

```
$('.btn').click((event) => {
  this.sendData()
})
```

No Skinny Arrow Function 😞

© NodeProgram.com, Node.University and Azat Mardan 2017

Note that you can mix and match (in one file) old `function` with `=>` in ES6 as you see fit. (Like most of ES6+ code.)

Good to know features of arrow functions

Implicit vs. Explicit Returns

Implicit means you don't type word `return` but the function becomes an expression (returns a value).

© NodeProgram.com, Node.University and Azat Mardan 2017

If you have *only* one line, then you can use implicit `return`.

If you have more than one line, then you must use explicit `return`.

© NodeProgram.com, Node.University and Azat Mardan 2017

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 + ' ' // always use explicit return
})
```

Now here's a 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
```

(Notice that I used the string templates?)

© NodeProgram.com, Node.University and Azat Mardan 2017

The parenthesis ( ) are optional for a single param

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?)

Aren't fat arrows are great? Use them.

For Of Comprehensions

© NodeProgram.com, Node.University and Azat Mardan 2017

Here is a problem with ES5: when we want to iterate over objects using its keys, we need to extract those keys first with `Object.keys()`

© NodeProgram.com, Node.University and Azat Mardan 2017

```
var books = ['Pro Express.js', 'React Quickly', 'Full Stack JavaScript']
books.author = 'Azat'
Object.keys(books).forEach(function (key) {
    console.log(books[key], key)
})
```

Output:

```
Pro Express.js 0
React Quickly 1
Full Stack JavaScript 2
Azat author
```

Meet `for of`

The for…of statement replaces standard for and forEach, and is similar to for…in except that that for…in iterates over keys and for…of over values.

for in is for keys/indices:

```
var books = ['Pro Express.js', 'React Quickly', 'Full Stack JavaScript']
for (let key in books){
  console.log(key) // indices: 0, 1, 2
}
```

for of is for values:

```
var books = ['Pro Express.js', 'React Quickly', 'Full Stack JavaScript']
for (let book of books){
  console.log(book) // values: Pro Express.js, React Quickly, Full Stack JavaScript
}
```

Differences

- for in work with object too, for of only with iterable objects such as Array,Map, Set, TypedArray, String, `arguments`
- for of will ignore any non-index properties, e.g., `book.author = 'Azat'`, while for in and forEach won't ignore them

```
var books = ['Pro Express.js', 'React Quickly', 'Full Stack JavaScript']
books.author = 'Azat'
for (let key in books){
  console.log(books[key]) // author: Azat
}

var books = ['Pro Express.js', 'React Quickly', 'Full Stack JavaScript']
books.author = 'Azat'
for (let value of books){
  console.log(value) // no author value Azat
}
```

My personal experience working with comprehensions is that they increase the code readability.

© NodeProgram.com, Node.University and Azat Mardan 2017

The End of Section 2 👍

© NodeProgram.com, Node.University and Azat Mardan 2017