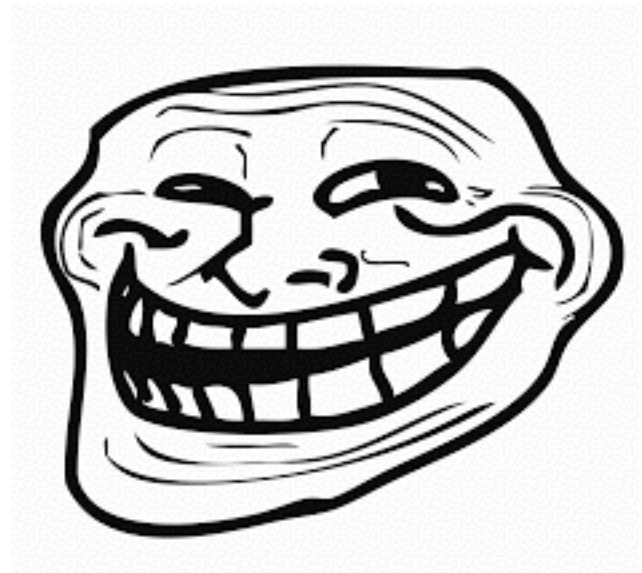
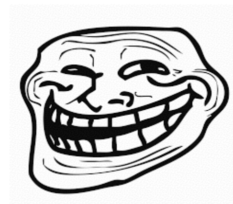
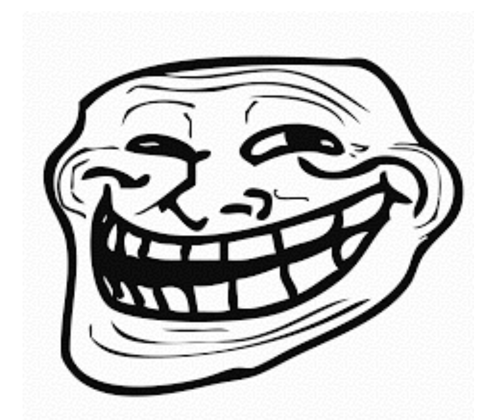
Aleksey Nudelman is WRONG!

Getting Started with ES6, Class free OOP, and Angular2



Just kidding. Not wrong. Just Different. Maybe…





Luis De Avila

Full-stack JavaScript

luis@ideanerd.com
www.ideanerd.com
www.linkedin.com/in/ldeavila

Overview

- Intro to best new, in my opinion, ES6 features
- Class free OOP
- Angular2 examples in ES6
- Tools for ES6 fun and profit
- My take on Angular2 with ES6

Key ES6 Features

- There are many new ES6 features
- Variable Declaration: **let**, **const**
- Destructuring: **{foo, bar} = params**
- Arrow functions: **() => {}**
- Modules: **export**, **import**

Variable Declaration

- In ES5 we have closure
- With ES6 we can now also have block-level scope
- **let** creates variables with block-level scope
- **const** same as let and locks reference to value

Variable Declaration

```
var foo = 'foo';

if(true){
  var bar = 'bar';
}

// logs 'foo' and 'bar'
console.log(foo, bar);
```

```
let foo = 'foo';

if(true){
  let bar = 'bar';
}

// logs 'foo' and generates reference
error
console.log(foo, bar);
```

```
var foo = {bar: 'bar'};

foo.baz = 'baz';

foo = {foo: ‘foo’};
```

```
const foo = {bar: 'bar'};

// This is ok.
foo.bar = 'bar';

// Causes error: 'read-only' error in
babel
foo = {bar: 'bar', baz:'baz'};
```

Destructuring

- Assign properties of an array/object to variables using syntax that looks similar to array/object literals.
- Provide default values

Destructuring

Old Way

```
function add(params) {
  // We provide 'default' for num1/num2 if
falsy
  var num1 = params.num1 || 0;
  var num2 = params.num2 || 0;

  return num1 + num2;
}
```

New Way

```
// destructure the params object into two
variables
function add(params) {
  var {num1 = 0, num2 = 0} = params;

  return num1 + num2;
}

////

// destructure inside the function
signature
function add({num1 = 0, num2 = 0} =
params) {
  return num1 + num2;
}
```

Arrow Functions

- In ES5 functions define their own **this** context
- In ES6 arrow functions offer a more concise syntax and inherit lexical **this** from the parent scope
- if no **{}**, then implicit return

Arrow functions

Old Way

```
$('.current-time').each(function () {
  // this refers to the DOM element
returned by JQuery
  var self = this;

  setInterval(function () {
    // if we used this inside the
callback, this would refer to the
callback's this.
    // We use self = this above as a way
to prevent the callback this 'shadowing'
the
    // DOM element this.
    $(self).text(Date.now());
  }, 1000);
});
```

New Way

```
$('.current-time').each(function () {
  // The arrow function does not create
its own this.
  setInterval(() => {
    $(this).text(Date.now())
  }, 1000);
});
```

Modules

- A common syntax for defining and loading modules
- Syntax similar to CommonJS (node.js) and supports async loading like AMD (Require.js)

Modules

Old Way: Module Pattern

```
// math exposed as a global variable
(function (window) {
  'use strict';

  // both add() and subtract() are
available
  var math = {
    VERSION: '1',
    add: function add(num1, num2) {
      return num1 + num2;
    },
    subtract: function subtract(num1,
num2) {
      return num1 - num2;
    }
  };

  window.math = math;

})(window);
```

New Way: Modules

```
let VERSION = '1';

function add() {
  return num1 + num2;
}

function subtract(num1, num2) {
  return num1 - num2;
}

// expose the variable and the functions
export {VERSION, add, subtract}
```

```
import * as math from 'math.js'
// Or only use what you want
//import {add} from 'math.js'

var total = math.add(1,1);
//var total = add(1,1);
```

How We Think of OOP Today

- Classical Inheritance
  - Have to do Classification Taxonomy
  - You'll likely get it wrong
  - Private data in ES6 classes not-ideal
- Prototypical Inheritance
  - Can be confusing
  - Retroactive inheritance

Class Free OOP

Factory Pattern

```
import {otherConstructor} from 'otherFile.js';

function constructor(spec) {
  // Use destructuring to create member variables
  let {member} = spec;

  // Can call another constructor and take/use what you
need
  let {other} = otherConstructor(spec);

  let method = function () {
    // have access to spec, member, other
  };

  // return an object that is an immutable public
interface
  return Object.freeze({
    method: method,
    other: other
  });
}

export {constructor}
```

Example

```
import {LogFirstLast} from ‘log-first-last.js’;

function LogNames(name) {
  // Use destructuring to create member variables
  let {first = 'John', last = 'Doe'} = name,

  // Can call another constructor and take/use what you
need
    {logFirst, logLast} = LogFirstLast(name),

    logFullName = function () {
      logFirst(first);
      logLast(last);
    };

  // return an object that is an immutable public
interface
  return Object.freeze({
    logFullName,
    logFirst,
    logLast
  });
}

export {LogNames}
```

Benefits/Drawbacks

- No drawbacks of classical or prototypical inheritance
- Easy private members; no Symbols, WeakMaps
- Clean, structured, readable code
- Takes a bit more memory but **who cares**

Angular2 in ES6

- Most Example are in Type Script
- Deconstruct what TS is doing and code it in ES6

Component

TypeScript

```
import {Component} from 'angular2/core';

@Component({
  selector: 'app',
  templateUrl: './app/app.component.html'
})

class App {
  constructor(){
    message: 'Canvas will go here.';
  }
}

export {app};
```

ES6

```
import {Component} from 'angular2/core';

let app = Component({
  selector: 'app'
})
  .View({
    templateUrl: './app/app.component.html'
  })
  .Class({
    constructor: function () {
      this.message = 'Canvas will go here.';
    }
  });

export {app};
```

ES6 Tools

- ES6 Module Loading & Package Management
- Linting

Loaders & Managers

- Many loader options: Require.js, LazyLoad
- Package managers: bower, npm
- I like JSPM… Why?

JSPM

- jspm is a package manager for the SystemJS universal module loader, built on top of the dynamic ES6 module loader
- Load any module format (**ES6, AMD, CommonJS and globals**) directly from any registry such as npm and GitHub with flat versioned dependency management. Any custom registry endpoints can be created through the Registry API.
- For development, load modules as separate files with ES6 and plugins compiled in the browser.
- For production (or development too), **optimize into a bundle, layered bundles or a self-executing bundle** with a single command.

Code Quality Tools

- Many linting, styling options: jsLint, jsHint, JSCS, esLint
- I like esLint… Why?

esLint

- **Very Flexible: All rules can be enabled/disabled/ tweaked**
- Easy to understand output
- Includes many rules not available in other linters
- **Best ES6 support**
- Supports custom reporters

.editorconfig

- Helps enforce code style/formatting across editors
- Supported by WebStorm, Sublime, Atom

What I learned

- ES6 here. Use it if you can to make code simpler
- Coding Angular2 in ES6 is for rebels. Learn TypeScript or be a rebel. Ideally, be both.
- Angular2 Testing needs work. Feels bloated.
- **Stay curious and have fun**

Thanks

luis@ideanerd.com
www.ideanerd.com
www.linkedin.com/in/ldeavila

References

- ES6 book: https://leanpub.com/exploring-es6
- Douglas Crockford Talk (Class free OOP): https://www.youtube.com/watch?v=PSGEjv3Tqo0
- Angular2: https://angular.io/
- Test Presentation by Julie Ralph: https://www.youtube.com/watch?v=C0F2E-PRm44
- Julie Ralph presto source: https://github.com/juliemr/ng2-test-seed
- Another sample tests: https://github.com/mgechev/angular2-seed
- JSPM: http://jspm.io/
- Linting Tools: http://www.sitepoint.com/comparison-javascript-linting-tools/
- esLint: http://eslint.org/