ECMASCRIPT 6

ORI CALVO

TRAINOLOGIC

Agenda

- Introduce interesting features in ECMAScript 6 and beyond

Bit Of History

1995	1997	1999	2009	2015	2016	2017
Developed at Netscape	ECMAScript 1	ES3	ES5	ES6	ES7	ES8
		Regex	Strict mode	A.K.A ECMAScript 2015		
		New control statements	JSON			
		try/catch	Properties			
			Maximum support			

Atwood's Law

Any application that can be written in JavaScript will eventually be written in JavaScript

Back to the future

… and, eventually, there will be no other language than JavaScript

ECMAScript Compatibility

- https://kangax.github.io/compat-table/es6/

- Chrome 67 – 98%
- Firefox 60 – 98%
- Edge 17 – 96%
- IE11 – 11% ☹

Supporting old Browsers

- Transpiling to the rescue …
- Write ES6 today
- Compile to ES5
- Run everywhere
- Popular transpilers
  - Typescript
  - Babel

Use Typescript

- Demo

Improving without breaking

- JavaScript does not complain about
  - Changing a string
  - Deleting global variable
  - Changing the value of undefined
- "use strict" (A.K.A strict mode)
  - Exception is thrown for all above scenarios

Block Scoped Variables

- What will be printed ?

```
var num = 11;

function run() {
    console.log(num);

    var num = 10;
}

run();
```

const/let don't hoist

- Same example as before

```
var num = 11;

function run() {
    console.log(num);

    const num = 10;
}

run();
```

Exception: num is not defined

let inside for loop

- What will be printed ?

```
function run() {
    for(var i=0; i<10; i++) {
        task(function() {
            console.log("Task #" + i + " completed");
        });
    }
}

function task(cb) {
    setTimeout(cb, 1000);
}

run();
```

And now ?

```
function run() {
    for(let i=0; i<10; i++) {
        task(function() {
            console.log("Task #" + i + " completed");
        });
    }
}

function task(cb) {
    setTimeout(cb, 1000);
}

run();
```

Use let instead of var

When do we use which ?

- const by default
- let otherwise
- var for legacy

Class – ES5

- The simple way

```
function Point(x, y) {
   function dump() {
      console.log(x + ", " + y);
   }

   function print(dx, dy) {
      x += dx;
      y += dy;
   }

   return {
      print: print,
      move: move,
   }
}

var pt1 = Point(5, 10);
var pt2 = Point(10, 20);

console.log(pt1 == pt2) // false
console.log(pt1.print == pt2.print) // false
```

x,y cannot be accessed from the outside world

dump, move are duplicated for every new Point object

Class – ES5 (prototype based)

```
function Point(x, y) {
    this.x = x;
    this.y = y;
}

Point.prototype.print = function() {
    console.log(this.x + ", " + this.y);
}

var pt1 = new Point(5, 10);
var pt2 = new Point(10, 20);

console.log(pt1 == pt2) // false
console.log(pt1.print == pt2.print) // true
```

Must use this to share data between constructor and prototype functions

Class – ES6

Syntactic sugar over prototype based code

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  print() {
    console.log(this.x + ", " + this.y);
  }
}

const pt1 = new Point(5, 10);
const pt2 = new Point(10, 20);

console.log(pt1.print == pt2.print) // true
```

Same design bug

- What will be printed ?

```
let nextTimerId = 1

class Timer {
  constructor(ms) {
    this.id = nextTimerId++;
    this.ms = ms;
  }

  start() {
    this.intervalId = setInterval(this.onTick, this.ms);
  }

  onTick() {
    console.log(this.id); // undefined
  }
}

const timer = new Timer(1000);
timer.start();
```

this is lost

this is window/undefined

Fat Arrow/Arrow Function

- Lexically captures this from surrounding context

```
let nextTimerId = 1

class Timer {
    constructor(ms) {
        this.id = nextTimerId++;
        this.ms = ms;
    }

    start() {
        this.intervalId = setInterval(() => this.onTick(), this.ms);
    }

    onTick() {
        console.log(this.id);
    }
}

const timer = new Timer(1000);
timer.start();
```

Challenge

- Let base class ctor change without breaking derived class

```
class Point {
   constructor(x,y) {
      this.x = x;
      this.y = y;
   }

   print() {
      console.log(this.x + ", " + this.y);
   }
}

class PointEx extends Point {
}

const pt = new PointEx(5, 10);
pt.print();
```

- But what if we want to run some code inside derived ctor ?

Rest Parameter

```
class PointEx extends Point {
  constructor(...args) {
    super(...args);

    console.log("derived");
  }
}

const pt = new PointEx(5, 10);
pt.print();
```

Read and send all parameters

Rest Parameter & Array

Prints 2, 3

```
const arr = [1,2,3];

const [num1, ...tail] = arr;

for(const num of tail) {
    console.log(num);
}
```

Can still use Class as a function

```
function profile(cls) {
    class ProfiledClass extends cls {
        constructor(... args) {
            super(... args);

            ++ProfiledClass.objectCount;
        }
    }

    ProfiledClass.objectCount = 0;

    return ProfiledClass;
}
```

```
const Point = profile(class {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }

    print() {
        console.log(this.x + ", " + this.y);
    }
});
```

```
const pt1 = new Point(5, 10);
const pt2 = new Point(10, 20);

console.log(Point.objectCount);
```

Strings

- JavaScript has no character data type
- Therefore we can use “ or ‘ to represent a string
- Starting ES6 we can use backtick `

```
function run() {
    const name = "Ori"
    const str = `Hello ${name}, how are you ?`;

    console.log(str);
}

run();
```

Default Parameter

- Boring …

```
function add(num1, num2 = 10) {
    return num1 + num2;
}

console.log(add(5));
```

Object Destructing

```
const contact = {
    id: 1,
    name: "Ori",
    email: "ori@trainologic.com",
};

const {email} = contact;

function log({id, name, email}) {
    console.log(id + ", " + name + ", " + email);
}

log(contact);
```

Array Destructing

- Same idea …

```
function getDetails() {
    const arr = [1, "Ori", "ori@trainologic.com"];
    return arr;
}

const [id, name, email] = getDetails();
console.log(id + ", " + name + ", email");
```

for ... of

- ES5 and below

```
const arr = ["A", "B", "C"];

for(let i=0; i<arr.length; i++) {
    console.log(arr[i]);
}

for(let index in arr) {
    console.log(index + ": " + arr[index]);
}

arr.forEach(function(value, index) {
    console.log(index + ": " + value);
});
```

- ES6

```
for(const value of arr) {
    console.log(value);
}
```

Iterable

- Any object can be iterated by for .. Of
- As long the object “implements” the Symbol.iterator behavior

```
const obj = {
    [Symbol.iterator]() {
        let next = 0;

        return {
            next: function() {
                return {
                    done: next == 10,
                    value: next++,
                };
            }
        }
    }
};

for(const val of obj) {
    console.log(val);
}
```

Generator

- Implement an iterable without messing with the Symbol.iterator behavior

```
function *getData() {
    for(let i=0; i<10; i++) {
        yield i;
    }
}

for(const num of getData()) {
    console.log(num);
}
```

Mapping Object to Object

- What is wrong ?

```
const ori = {
    id: 1,
    name: "Ori"
};

const roni = {
    id: 2,
    name: "Roni"
};

const map = {};
map[ori] = true;

console.log(map[roni]);
```

Better Solution

Need to implement getHashCode by modifying the incoming object

```
const ori = {
    id: 1,
    name: "Ori"
};

const roni = {
    id: 2,
    name: "Roni"
};

const map = {};
map[getHashCode(ori)] = true;

console.log(map[getHashCode(roni)]);
```

getHashCode

- Tricky – Can’t do that in C#/Java

```
const getHashCode = (function() {
    let nextHash = 1;
    const MAGIC_FIELD = "##magic_field##";

    function getHashCode(obj) {
        let hash = obj[MAGIC_FIELD];
        if(!hash) {
            hash = obj[MAGIC_FIELD] = nextHash++;
        }

        return hash;
    }

    return getHashCode;
})();
```

ES6 Map

- No need for ugly tricks any more
- Any object can be used as a key

```
const map = new Map();

const ori = {
    id: 1,
    name: "Ori",
}

map.set(ori, 1);

const likeOri = {
    id: 1,
    name: "Ori",
}

console.log(map.has(likeOri)); // false
```

WeakMap

- Imagine an infrastructure that needs to attach additional information for every application’s object
- We don’t want to modify the object
- We can use a Map
- However this means that the infrastructure holds application’s objects alive
- Use WeakMap instead

WeakMap

```
const map = new WeakMap();

class Contact {
    constructor(name) {
        this.name = name;
    }
}

let ori = new Contact("Ori");
let roni = new Contact("Roni");
map.set(ori, roni);

setTimeout(function() {
    ori = null;
    console.log("Ori can now be GC'ed");
}, 1000);
```

Modules

- At the beginning there was a pattern

```
const network = (function() {
    let lastRequestTime;
    let requestCount;

    function get() {
    }

    function post() {
    }

    function put() {
    }

    return {
        get,
        last,
        put,
    };
})();
```

Data is encapsulated and cannot be accessed by external code

CommonJS

- The CommonJS initiative has its own way

```
function doSomething() {
   console.log("lib");
}

exports.doSomething = doSomething;
```

```
const lib = require("./lib");

lib.doSomething();
```

- But no browser implemented that specification

AMD

- AMD does not require browser help
- It is implement by the require.js library

- Too much details … take me to the interesting part

ES6 Modules

- Standard ECMAScript syntax based on import/export keywords
- Experimental status under NodeJS

```
export function run() {
    console.log("run")
}
```

```
import {run} from "./lib.js";

run();
```

Tree Shaking

- Smart bundlers like Rollup and Webpack can "shake" your code and keep only relevant parts

```
export function run1() {
    console.log("run1");
}

export function run2() {
    console.log("run2");
}
```

```
import {run1} from "./lib";

run1();
```

Generated bundle does not include function run2

```
(function () {
    'use strict';

    function run1() {
        console.log("run1");
    }

    run1();

}());
```

Summary

- const/let
- Fat arrow
- class syntax
- Destructing object/array
- ES6 Modules
- Collections