

ECMAScript 6

ORI CALVO

TRAINOLOGIC



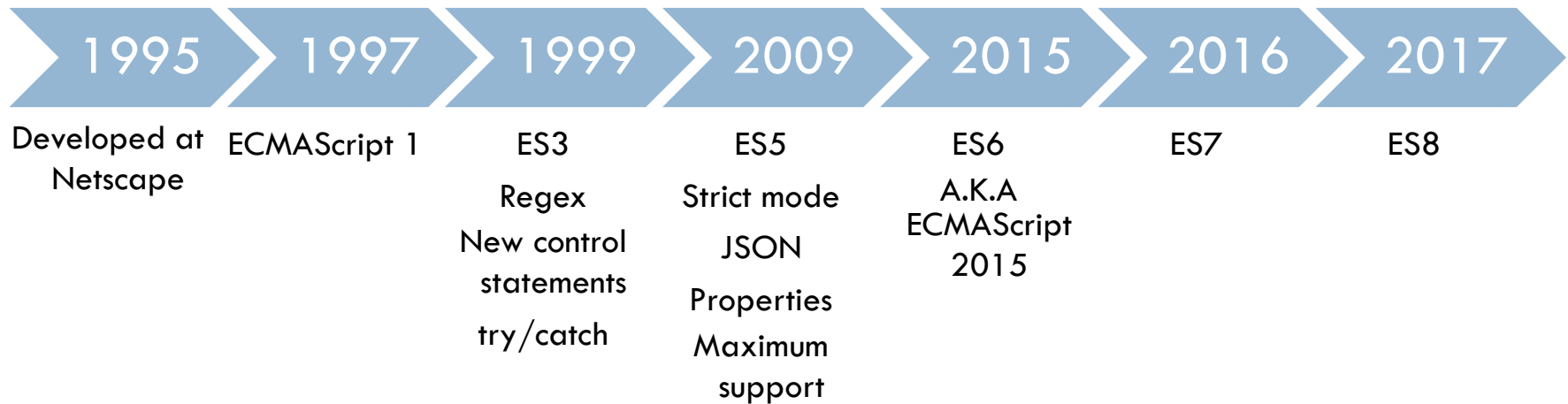
Agenda

2

- Analyze interesting features in ECMAScript 6 and beyond

Bit Of History

3



Atwood's Law

4

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

Back to the future

5

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

ECMAScript Compatibility

6

- <https://kangax.github.io/compat-table/es6/>
- Chrome 67 – 98%
- Firefox 60 – 98%
- Edge 17 – 96%
- IE11 – 11% ☹️

Supporting old Browsers

7

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

Use Typescript

8

□ Demo

Improving without breaking

9

- 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

10

- What will be printed ?

```
var num = 11;

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

  var num = 10;
}

run();
```

const/let don't hoist

11

- Same example as before

Exception:
num is not
defined

```
var num = 11;

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

  const num = 10;
}

run();
```

let inside for loop

12


□ 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 ?

13

Use let
instead of
var



```
function run() {  
  for(let i=0; i<10; i++) {  
    task(function() {  
      console.log("Task #" + i + " completed");  
    });  
  }  
}  
  
function task(cb) {  
  setTimeout(cb, 1000);  
}  
  
run();
```

When do we use which ?

14

- ❑ `const` by default
- ❑ `let` otherwise
- ❑ `var` for legacy

Class – ES5

15

□ The simple way

x,y cannot be
accessed
from the
outside world

```
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
```

dump, move are
duplicated for
every new Point
object

Class – ES5 (prototype based)

16

Must use this to
share data
between
constructor and
prototype
functions

```
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
```


Class – ES6

17

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

18

□ 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
window/undefined

this is
lost

Fat Arrow/Arrow Function

19

- 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

20

- 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

21

Read and
send all
parameters

```
class PointEx extends Point {  
  constructor(...args) {  
    super(...args);  
  
    console.log("derived");  
  }  
}  
  
const pt = new PointEx(5, 10);  
pt.print();
```

Rest Parameter & Array

22

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

23

```
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

24

- ❑ 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

25

□ Boring ...

```
function add(num1, num2 = 10) {  
  return num1 + num2;  
}  
  
console.log(add(5));
```

Object Destructuring

26

```
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 Destructuring

27

- 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

28

□ 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

29

- 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

30

- 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);  
}
```

Async Iterator (ES2018)

31

```
async function *getData() {  
  await delay(1000);  
  yield 1;  
  await delay(2000);  
  yield 2;  
}
```

```
async function main() {  
  for await(const line of getData()) {  
    console.log(line);  
  }  
}
```

Mapping Object to Object

32

□ 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

33

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

34

□ 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

35

- ❑ 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

36

- 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

37

```
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);

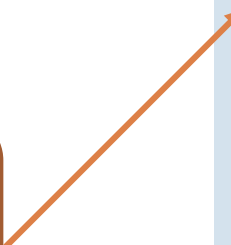
setTimeout(function() {
  ori = null;
  console.log("Ori can now be GC'ed");
}, 1000);
```

Modules

38

- At the beginning there was a pattern

Data is
encapsulated
and cannot be
accessed by
external code



```
const network = (function() {  
  let lastRequestTime;  
  let requestCount;  
  
  function get() {  
  }  
  
  function post() {  
  }  
  
  function put() {  
  }  
  
  return {  
    get,  
    last,  
    put,  
  };  
})();
```

CommonJS

39

- 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

40

- AMD does not require browser help
- It is implement by the **require.js** library
- Too much details ... take me to interesting part

ES6 Modules

41

- 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

42

- 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();  
  
})();
```

Promise

43

- Promise object represent an asynchronous operation
- A function the completes in the future returns a promise
- The caller keeps the promise object and uses the then/catch handlers
- Same idea as
 - ▣ .NET Task
 - ▣ Java Future

Promise

44

```
function main() {
  delay(1500).then(() => {
    console.log("OK");
  }).catch(() => {
    console.log("FAIL");
  });
}

function delay(ms) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve();
    }, ms);
  });
}
```

Chaining Promises

45

- The caller can easily create a new promise with different resolved value

```
function main() {  
  waitAndReturn42(1500).then(res => {  
    console.log(res);  
  });  
}  
  
function waitAndReturn42() {  
  return delay(1500).then(_ => 42);  
}
```

Nesting

46

- Although being a significant improvement over callbacks
- Promise based code is not as simple as blocking/plain code

```
function main() {  
  task1().then(res => {  
    if(res == 42) {  
      return task2().then(() => {  
  
        console.log("COMPLETED");  
      });  
    }  
    else {  
      console.log("COMPLETED");  
    }  
  });  
}
```

```
function plainSynchronousCode() {  
  if(task1() == 42) {  
    task2();  
  }  
  
  console.log("COMPLETED");  
}
```

async/await

47

- Any function can be awaited
- If the function returns a promise → Browser uses then/catch
- If function returns non promise → Browser does nothing

```
async function main() {  
  if(await task1() == 42) {  
    await task2();  
  }  
  
  console.log("COMPLETED");  
}
```

Handling Errors

48

- We can use try/catch 😊

```
async function main() {  
  try {  
    await task();  
  }  
  catch(err) {  
    console.log("ERROR: " + err.message);  
  }  
}  
  
async function task() {  
  await delay(1500);  
  throw new Error("Ooops");  
}
```


Typescript Specifics

49

- Types
- Interface/implementations
- Accessibility
- Enum
- Generic
- Static
- Abstract
- Constructor Assignment

THANK YOU !

