ECMAScript 5 and 6

The present and future of JavaScript

Tom Van Cutsem


@tvcutsem

My involvement in JavaScript

- 2004-2008: built up expertise in (dynamic) programming languages research during my PhD
- 2010: Visiting Faculty at Google, joined Caja team
- Joined ECMA TC39 (Javascript standardization committee)

- Actively contributed to the ECMAScript 6 specification

Talk Outline

- Part I: the past and present of ECMAScript
- Part II: the future of ECMAScript

Part I: the past and present of ECMAScript

JavaScript's origins

- Invented by Brendan Eich in 1995, then an intern at Netscape, to support client-side scripting in Netscape navigator
- First called *LiveScript*, then *JavaScript*, then standardized as *ECMAScript*
- Microsoft "copied" JavaScript in IE JScript, "warts and all"

Netscape®

What developers think about JavaScript

- Lightning talk Gary Bernhardt at CodeMash 2012
- https://www.destroyallsoftware.com/talks/wat

The world’s most misunderstood language

See also: “JavaScript: The World's Most Misunderstood Programming Language” by Doug Crockford at http://www.crockford.com/javascript/javascript.html

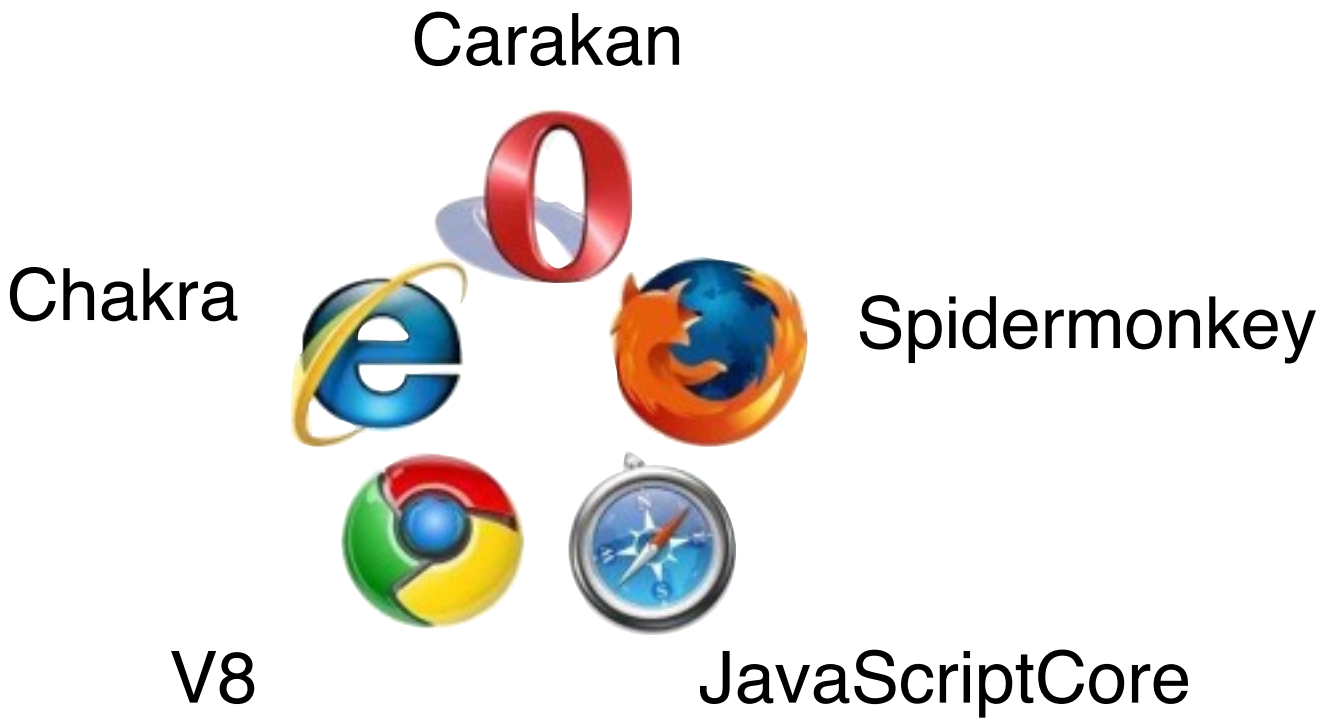
The Good Parts

- Functions as first-class objects
- Dynamic objects with prototypal inheritance
- Object literals
- Array literals

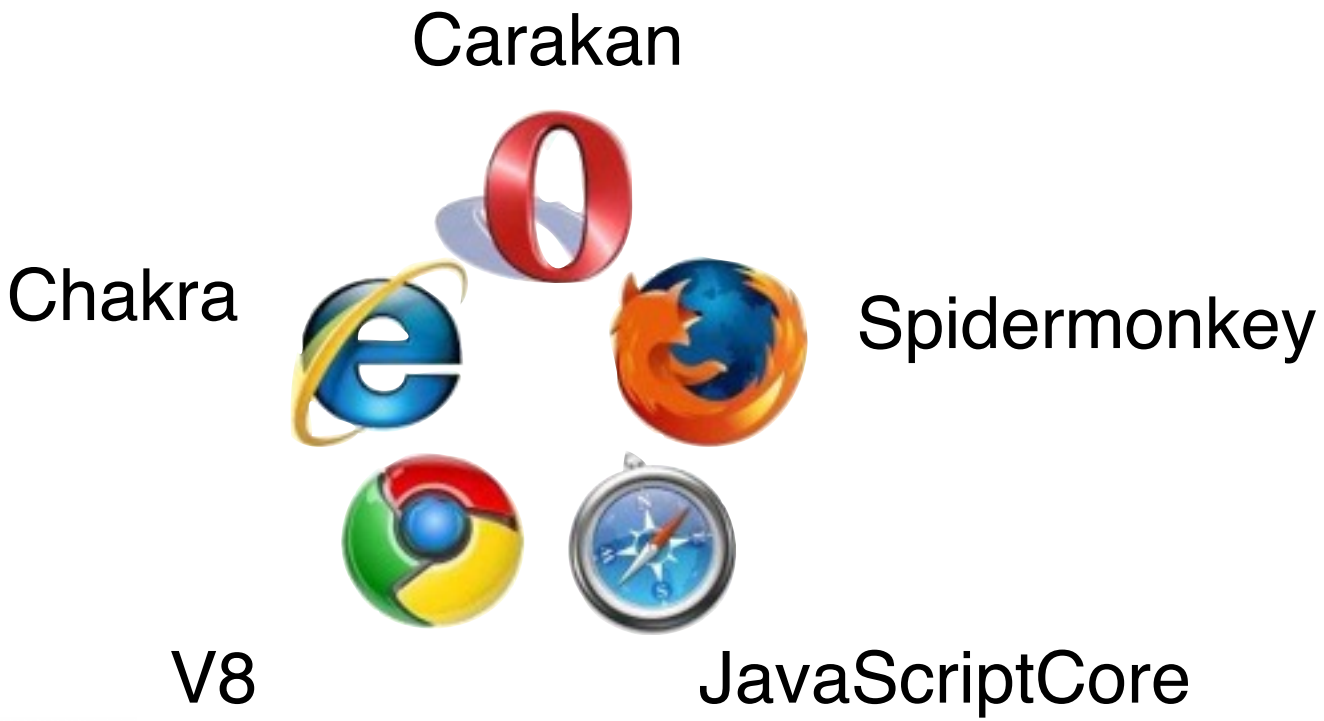
The Bad Parts

- Global variables (no modules)
- Var hoisting (no block scope)
- `with` statement
- Implicit type coercion
- …

ECMAScript: “Standard” JavaScript

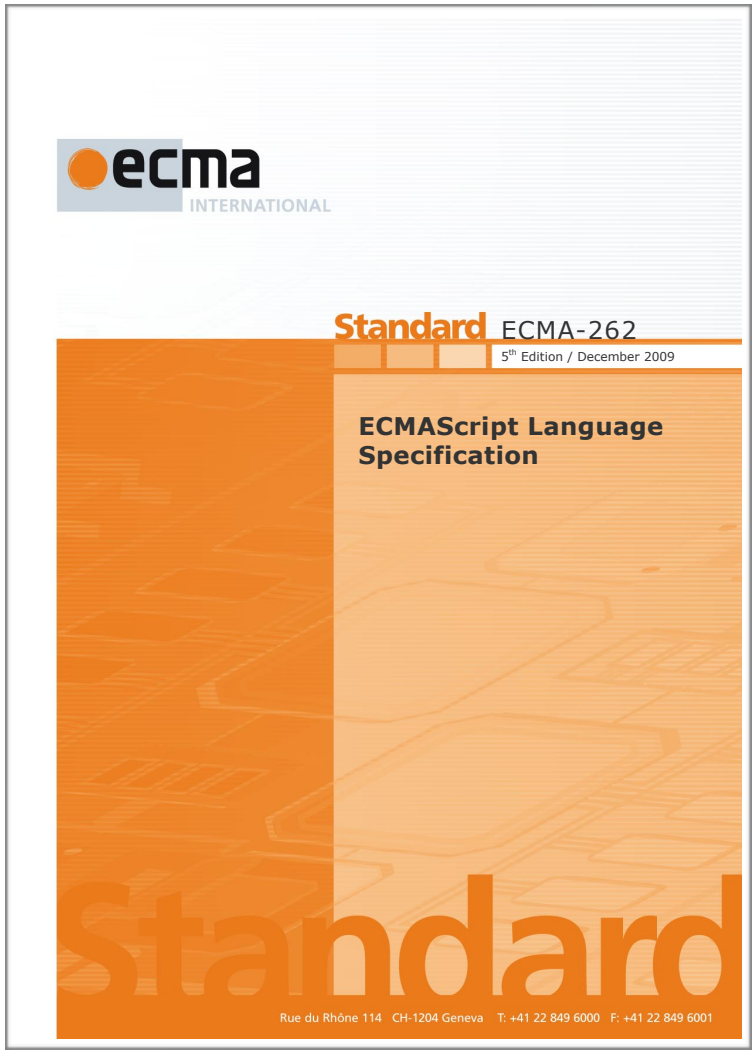


ECMAScript: “Standard” JavaScript



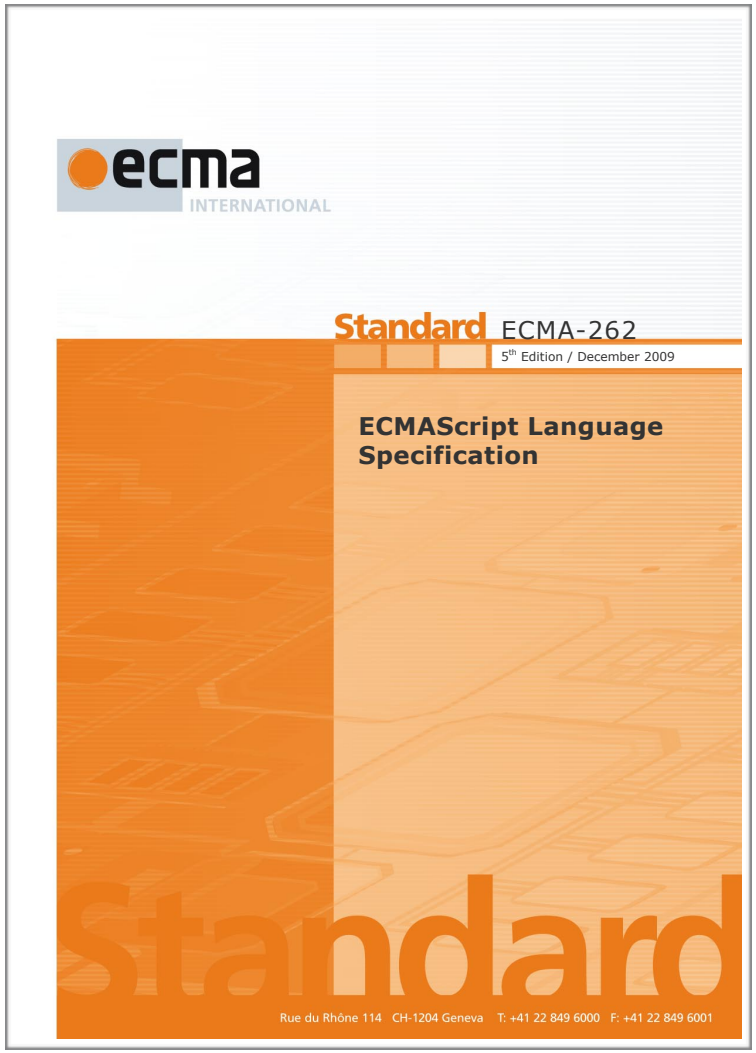
ECMAScript specification

- 1st ed. 1997
- 2nd ed. 1998
- 3rd ed. 1999
- ~~4th ed.~~
- 5th ed. 2009
- *6th ed. end 2014 / mid 2015*



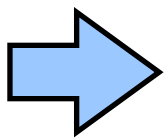
ECMAScript specification

- 1st ed. 1997
- 2nd ed. 1998
- 3rd ed. 1999
- ~~4th ed.~~
- **5th ed. 2009**
- *6th ed. end 2014 / mid 2015*



ECMAScript 5

- Many new standard API methods, e.g. Array map, filter, forEach, etc.
- Built-in support for parsing/generating JSON

~~`eval(jsonString)`~~  `JSON.parse(jsonString)`

- Ability to make properties of objects immutable (Object.freeze)
- Strict mode

Ecmascript 5 Strict mode

- Safer, more robust, subset of the language
- Why?
  - No silent errors
  - True static scoping rules
  - No global object leakage

Ecmascript 5 Strict mode

- Explicit opt-in to avoid backwards compatibility constraints
- How to opt-in
  - Per “program” (file, script tag, ...)
  - Per function
- Strict and non-strict mode code can interact (e.g. on the same web page)

```
<script>
"use strict";
...
</script>
```

```
function f() {
  "use strict";
  ...
}
```

Static scoping in ES5

- ECMAScript 5 non-strict is not statically scoped
- Four violations:
  - `with (obj) { x }` statement
  - `delete x;` // may delete a statically visible var
  - `eval('var x = 8');` // may add a statically visible var
  - Assigning to a non-existent variable creates a new global variable

    ```
    function f() { var xfoo; xFoo = 1; }
    ```

Ecmascript 5 Strict: syntactic restrictions

- The following are forbidden in strict mode (signaled as syntax errors):

```
with (expr) {
  ...x...
}
```

```
{ a: 1,
  b: 2,
  b: 3 } // duplicate property
```

```
function f(a,b,b) {
  // repeated param name
}
```

```
delete x; // deleting a variable
```

```
if (a < b) {
  // declaring functions in blocks
  function f(){}
}
```

```
var n = 023; // octal literal
```

```
function f(eval) {
  // eval as variable name
}
```

Ecmascript 5 Strict

- Runtime changes (fail silently outside of strict mode, throw an exception in strict mode)

```
function f() {
  "use strict";
  var xfoo;
  xFoo = 1;  // error: assigning to an undeclared variable
}
```

```
"use strict";
var p = Object.freeze({x:0,y:0});
delete p.x; // error: deleting a property from a frozen object
```

Ecmascript 5 Strict: avoid global object leakage

- Runtime changes: default `this` bound to `undefined` instead of the global object

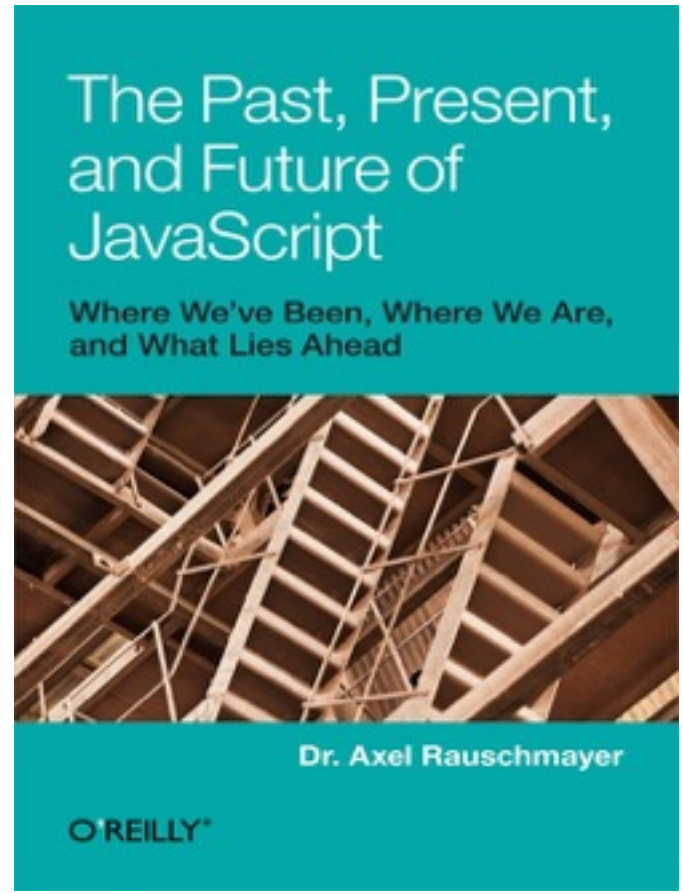
```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

var p = new Point(1,2);
var p = Point(1,2);
// window.x = 1;
// window.y = 2;
print(x) // 1 (bad!)
```

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

var p = new Point(1,2);
var p = Point(1,2);
// undefined.x = 1;
// error (good!)
```

Part II: the future of ECMAScript

ECMAScript 6

- Major update: many new features (too many to list here *)
- Will focus on three loose themes:
  - Improving functions
  - Improving modularity
  - Improving control flow



* see https://github.com/lukehoban/es6features for an overview of ES6 features

ECMAScript 6: improving functions

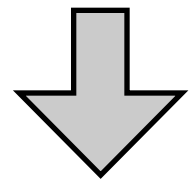
- Arrow functions
- Rest arguments
- Optional arguments
- Multiple return values and destructuring

ECMAScript 6: arrow functions

- Shorter, and also automatically captures current value of `this`

ES5

```
function sum(array) {
  return array.reduce(
    function(x, y) { return x + y; }, 0);
}
```



ES6

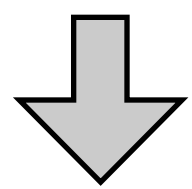
```
function sum(array) {
  return array.reduce((x, y) => x + y, 0);
}
```

ECMAScript 6: arrow functions

- Shorter, and also automatically captures current value of `this`

ES5

```
function sum(array) {
  return array.reduce(
    function(x, y) { return x + y; }, 0);
}
```



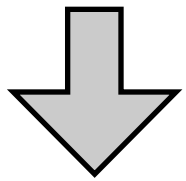
ES6

```
function sum(array) {
  return array.reduce((x, y) => x + y, 0);
}
```

ECMAScript 6: rest arguments

ES5

```
function printf(format) {
  var rest = Array.prototype.slice.call(arguments,1);
  …
}
```



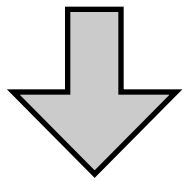
ES6

```
function printf(format, ...rest) {
  …
}
```

ECMAScript 6: rest arguments

ES5

```
function printf(format) {
  var rest = Array.prototype.slice.call(arguments,1);
  …
}
```



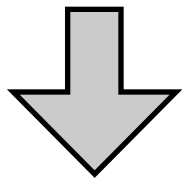
ES6

```
function printf(format, ...rest) {
  …
}
```

ECMAScript 6: optional arguments

ES5

```
function greet(arg) {
  var name = arg || “world”;
  return “Hello, ” + name;
}
```



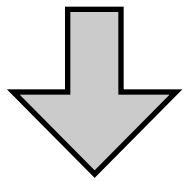
ES6

```
function greet(name = “world”) {
  return “Hello, ” + name;
}
```

ECMAScript 6: optional arguments

ES5

```
function greet(arg) {
  var name = arg || “world”;
  return “Hello, ” + name;
}
```



ES6

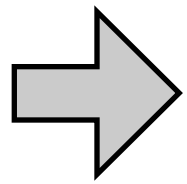
```
function greet(name = “world”) {
  return “Hello, ” + name;
}
```

ECMAScript 6: destructuring

```
// div(a,b) = q,r <=> a = q*b + r
function div(a, b) {
  var quotient = Math.floor(a / b);
  var remainder = a % b;
  return [quotient, remainder];
}
```

ES5

```
var result = div(4, 3);
var q = result[0];
var r = result[1];
```



ES6

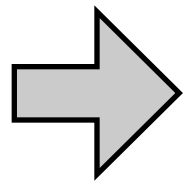
```
var [q,r] = div(4, 3);
```

ECMAScript 6: destructuring

```
// div(a,b) = q,r <=> a = q*b + r
function div(a, b) {
  var quotient = Math.floor(a / b);
  var remainder = a % b;
  return [quotient, remainder];
}
```

ES5

```
var result = div(4, 3);
var q = result[0];
var r = result[1];
```



ES6

```
var [q,r] = div(4, 3);
```

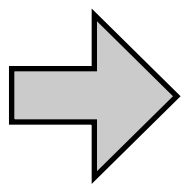
ECMAScript 6: improving modularity

- Classes (with single-inheritance)
- Modules

ECMAScript 6: classes

- All code inside a class is implicitly opted into strict mode!

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

Point.prototype = {
  toString: function() {
    return "[Point...]";
  }
}

var p = new Point(1,2);
p.x;
p.toString();
```

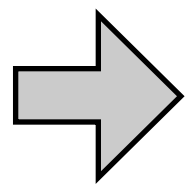


```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  toString() {
    return "[Point...]";
  }
}

var p = new Point(1,2);
p.x;
p.toString();
```

ECMAScript 6: classes

- All code inside a class is implicitly opted into strict mode!

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

Point.prototype = {
  toString: function() {
    return "[Point...]";
  }
}

var p = new Point(1,2);
p.x;
p.toString();
```

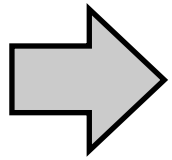


```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  toString() {
    return "[Point...]";
  }
}

var p = new Point(1,2);
p.x;
p.toString();
```

ECMAScript 6: modules

- All code inside a module is implicitly opted into strict mode!

```
<script>
var x = 0; // global
var myLib = {
  inc: function() {
    return ++x;
  }
};
</script>
```



```
<script type="module"
        name="myLib">
var x = 0; // local!
export function inc() {
  return ++x;
}
</script>
```

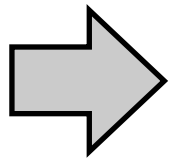
```
<script>
var res = myLib.inc();
</script>
```

```
<script type="module">
import { inc } from 'myLib';
var res = inc();
</script>
```

ECMAScript 6: modules

- All code inside a module is implicitly opted into strict mode!

```
<script>
var x = 0; // global
var myLib = {
  inc: function() {
    return ++x;
  }
};
</script>
```



```
<script type="module"
        name="myLib">
var x = 0; // local!
export function inc() {
  return ++x;
}
</script>
```

```
<script>
var res = myLib.inc();
</script>
```

```
<script type="module">
import { inc } from 'myLib';
var res = inc();
</script>
```

ECMAScript 6: improving control flow

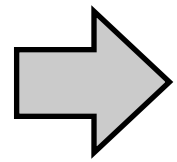
- Iterators
- Generators
- Promises
- (async/await)

ECMAScript 6 Iterators

```
function fibonacci() {
  var pre = 0, cur = 1;
  return {
    next: function() {
      var temp = pre;
      pre = cur;
      cur = cur + temp;
      return { done: false, value: cur }
    }
  }
}
```

ES5

```
var iter = fibonacci();
var nxt = iter.next();
while (!nxt.done) {
  var n = nxt.value;
  if (n > 100)
    break;
  print(n);
  nxt = iter.next();
}
```



ES6

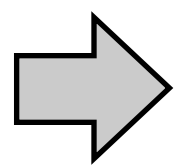
```
for (var n of fibonacci) {
  if (n > 100)
    break;
  print(n);
}
```

// generates 1, 1, 2, 3, 5, 8, 13, 21, …

ECMAScript 6 Iterators

```
function fibonacci() {
  var pre = 0, cur = 1;
  return {
    next: function() {
      var temp = pre;
      pre = cur;
      cur = cur + temp;
      return { done: false, value: cur }
    }
  }
}
```

```
var iter = fibonacci();
var nxt = iter.next();
while (!nxt.done) {
  var n = nxt.value;
  if (n > 100)
    break;
  print(n);
  nxt = iter.next();
}
```



```
for (var n of fibonacci) {
  if (n > 100)
    break;
  print(n);
}
```

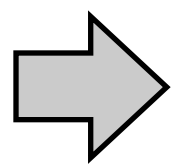
// generates 1, 1, 2, 3, 5, 8, 13, 21, …

ECMAScript 6 Generators

- A generator function implicitly creates and returns an iterator

ES5

```
function fibonacci() {
  var pre = 0, cur = 1;
  return {
    next: function() {
      var tmp = pre;
      pre = cur;
      cur = cur + tmp;
      return { done: false, value: cur }
    }
  }
}
```



ES6

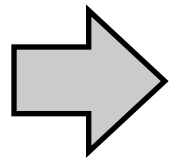
```
function* fibonacci() {
  var pre = 0, cur = 1;
  for (;;) {
    var tmp = pre;
    pre = cur;
    cur = cur + tmp;
    yield cur;
  }
}
```

ECMAScript 6 Generators

- A generator function implicitly creates and returns an iterator

ES5

```
function fibonacci() {
  var pre = 0, cur = 1;
  return {
    next: function() {
      var tmp = pre;
      pre = cur;
      cur = cur + tmp;
      return { done: false, value: cur }
    }
  }
}
```



ES6

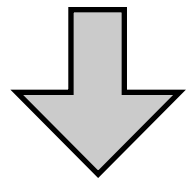
```
function* fibonacci() {
  var pre = 0, cur = 1;
  for (;;) {
    var tmp = pre;
    pre = cur;
    cur = cur + tmp;
    yield cur;
  }
}
```

ECMAScript 6 Promises

- A promise is a placeholder for a value that may only be available in the future

ES5

```
readFile("hello.txt", function (err, content) {
  if (err) {
    // handle error
  } else {
    // use content
  }
})
```



ES6

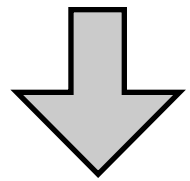
```
var pContent = readFile("hello.txt");
pContent.then(function (content) {
  // use content
}, function (err) {
  // handle error
});
```

ECMAScript 6 Promises

- A promise is a placeholder for a value that may only be available in the future

ES5

```
readFile("hello.txt", function (err, content) {
  if (err) {
    // handle error
  } else {
    // use content
  }
})
```



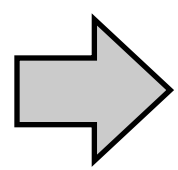
ES6

```
var pContent = readFile("hello.txt");
var p2 = pContent.then(function (content) {
  // use content
}, function (err) {
  // handle error
});
```

ECMAScript 6 Promises

- Promises can be *chained* to avoid callback hell

```
// step2(value, callback) -> void

step1(function (value1) {
    step2(value1, function(value2) {
        step3(value2, function(value3) {
            step4(value3, function(value4) {
                // do something with value4
            });
        });
    });
});
```



```
// promisedStep2(value) -> promise

Q.fcall(promisedStep1)
.then(promisedStep2)
.then(promisedStep3)
.then(promisedStep4)
.then(function (value4) {
    // do something with value4
})
.catch(function (error) {
    // handle any error here
})
.done();
```

Example adapted from https://github.com/kriskowal/q

ECMAScript 6 Promises

- Promises already exist as a library in ES5
- Personal favorite: Q (cf. https://github.com/kriskowal/q)

```
npm install q
```

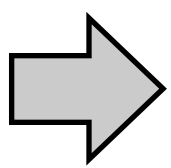
- Then why standardize?
  - Wide disagreement on a single Promise API. ES6 settled on an API called “Promises/A+”. See promisesaplus.com
  - Standard API allows platform APIs to use Promises as well
  - W3C’s latest DOM APIs already use promises

ECMAScript **7**: async/await

- async/await is a C# 5.0 feature that enables asynchronous programming using “direct style” control flow (i.e. no callbacks)

ES6

```
// promisedStep2(value) -> promise

Q.fcall(promisedStep1)
.then(promisedStep2)
.then(promisedStep3)
.then(promisedStep4)
.then(function (value4) {
    // do something with value4
})
.catch(function (error) {
    // handle any error here
})
.done();
```



ES7

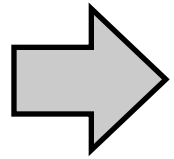
```
// step2(value) -> promise

(async function() {
  try {
    var value1 = await step1();
    var value2 = await step2(value1);
    var value3 = await step3(value2);
    var value4 = await step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
}())
```

async/await in ECMAScript **6**

- Generators can be used as async functions, with some tinkering
- E.g. using Q in node.js (>= 0.11.x with `--harmony` flag)

ES7

```
(async function() {
  try {
    var value1 = await step1();
    var value2 = await step2(value1);
    var value3 = await step3(value2);
    var value4 = await step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
}())
```



ES6

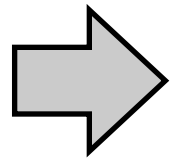
```
Q.async(function*() {
  try {
    var value1 = yield step1();
    var value2 = yield step2(value1);
    var value3 = yield step3(value2);
    var value4 = yield step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
})()
```

async/await in ECMAScript **6**

- Generators can be used as async functions, with some tinkering
- E.g. using Q in node.js (>= 0.11.x with `--harmony` flag)

ES7

```
(async function() {
  try {
    var value1 = await step1();
    var value2 = await step2(value1);
    var value3 = await step3(value2);
    var value4 = await step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
}())
```



ES6

```
Q.async(function*() {
  try {
    var value1 = yield step1();
    var value2 = yield step2(value1);
    var value3 = yield step3(value2);
    var value4 = yield step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
})()
```

ECMAScript 6: timeline

- Current draft is nearly feature-complete. Available online: http://people.mozilla.org/~jorendorff/es6-draft.html
- Spec needs to be ratified by ECMA, may take up to mid-2015
- However: browsers will not support ES6 overnight
- Parts of ES6 already supported on some browsers today*
- Use “transpilers” in the meantime to bridge the ES5-ES6 gap

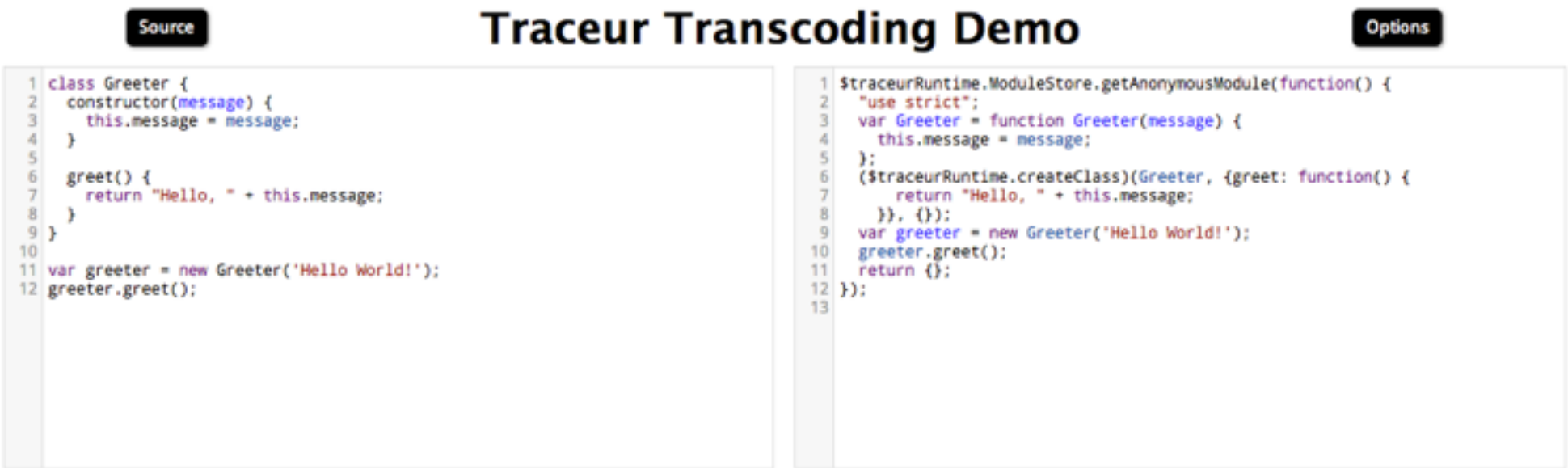
* see http://kangax.github.io/es5-compat-table/es6/ for current compatibility status

ECMAScript 6 transpilers: Traceur Compiler

- Google's ES6 to ES5 compiler
- https://github.com/google/traceur-compiler
- Installation: `npm install -g traceur`
- Usage: `traceur --script es6source.js --out es5source.js`

Traceur Compiler: Demo

- Demo: http://google.github.io/traceur-compiler/demo/repl.html

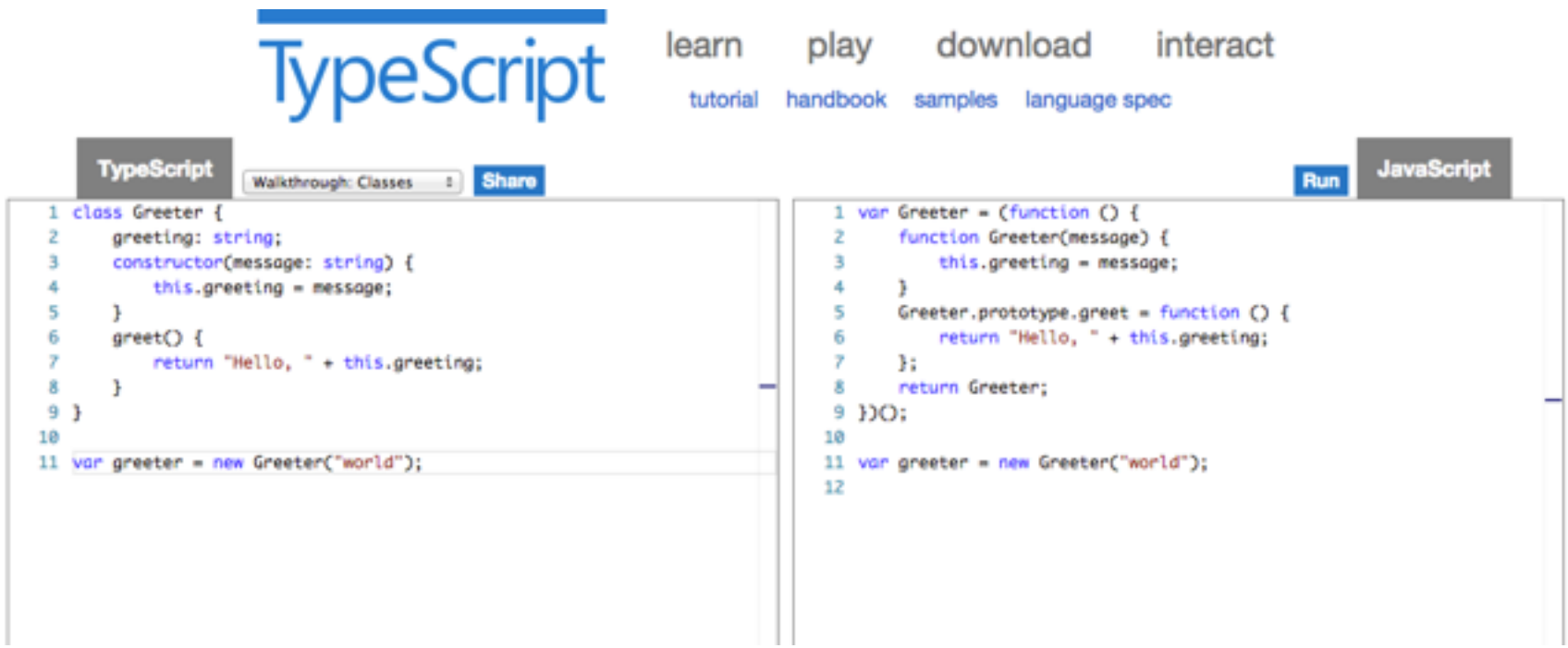


TypeScript

- Technically not an ES6 transpiler, but a new language from Microsoft with the aim of being roughly a superset of ES6
- Can use classes, modules and arrow functions today in TypeScript
- Bonus: type inference

TypeScript: Demo

- Demo: http://www.typescriptlang.org/Playground/

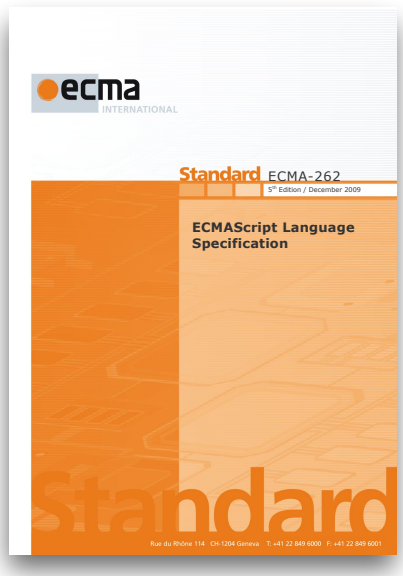


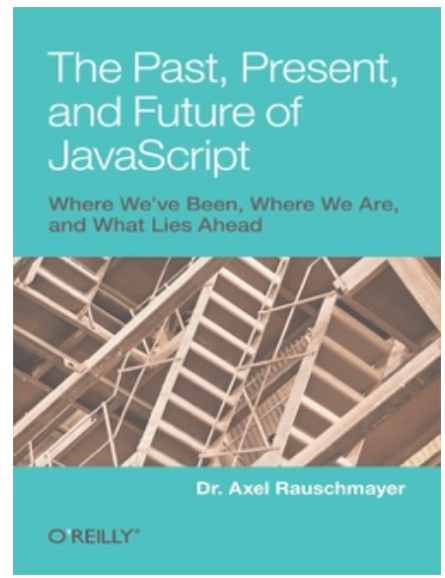
Wrap-up

Take-home messages

- Strict mode: a saner basis for the future evolution of JavaScript
- Opt-in subset that removes some of JavaScript's warts. Use it!

ECMAScript 5

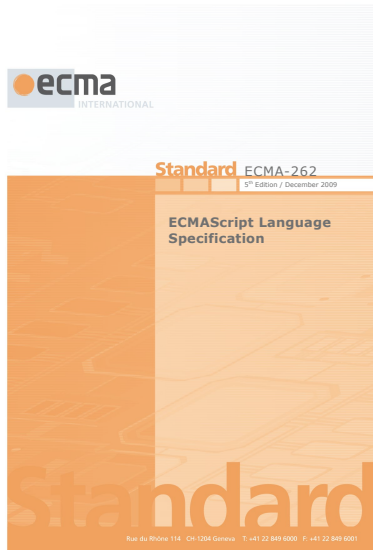


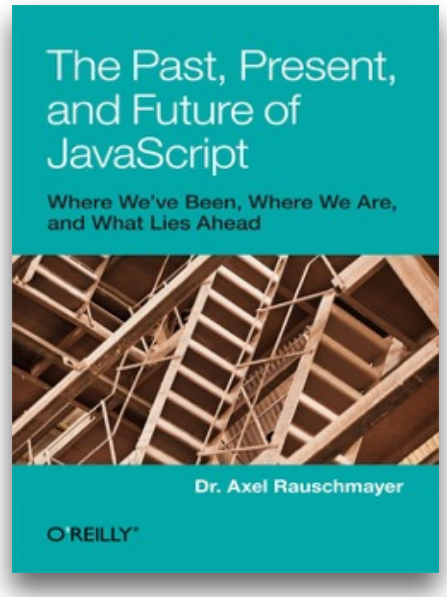


Take-home messages

- ECMAScript 6 is a *major* upgrade to the language
- Expect a gradual upgrade path and use transpilers to bridge the gaps

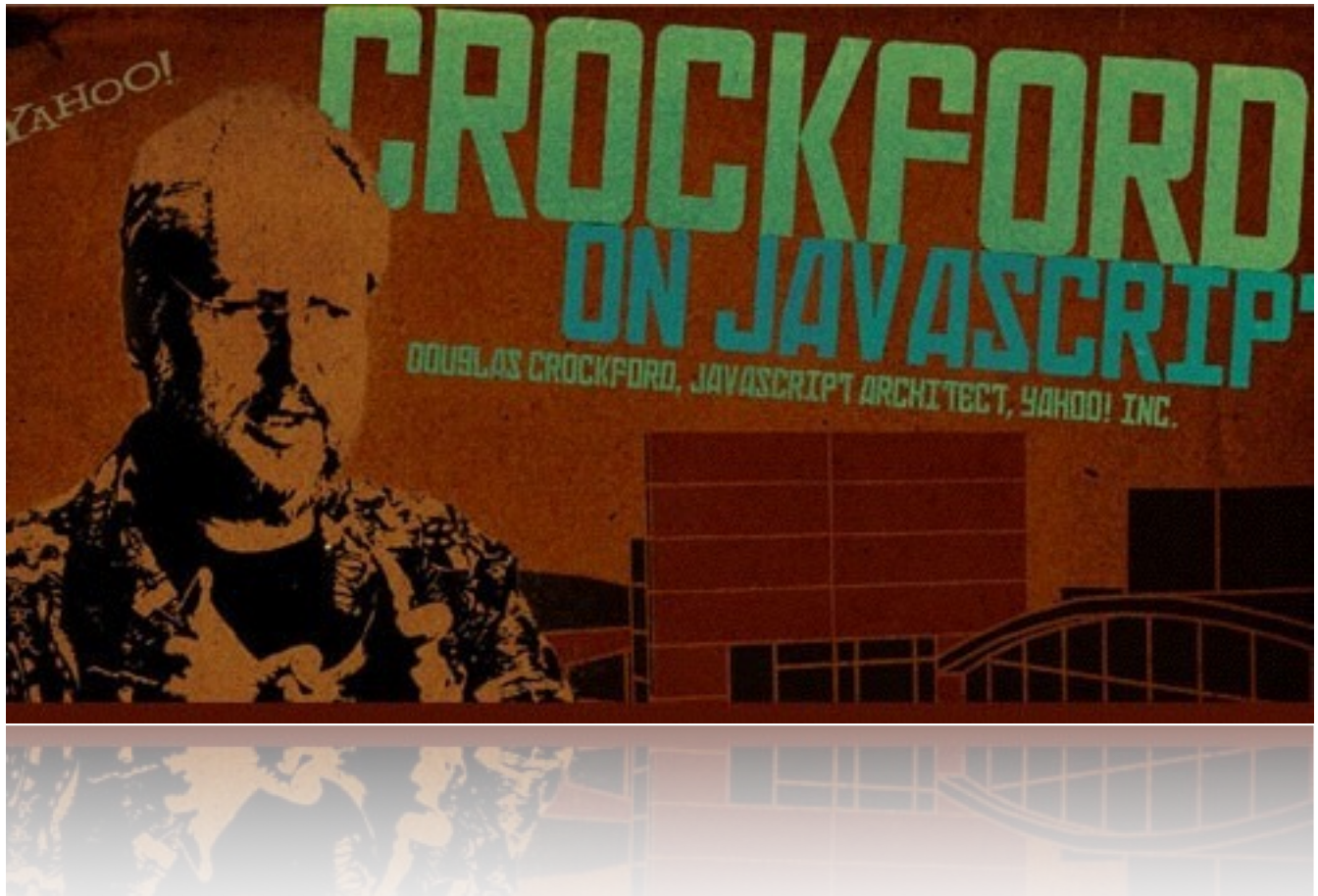
ECMAScript 6



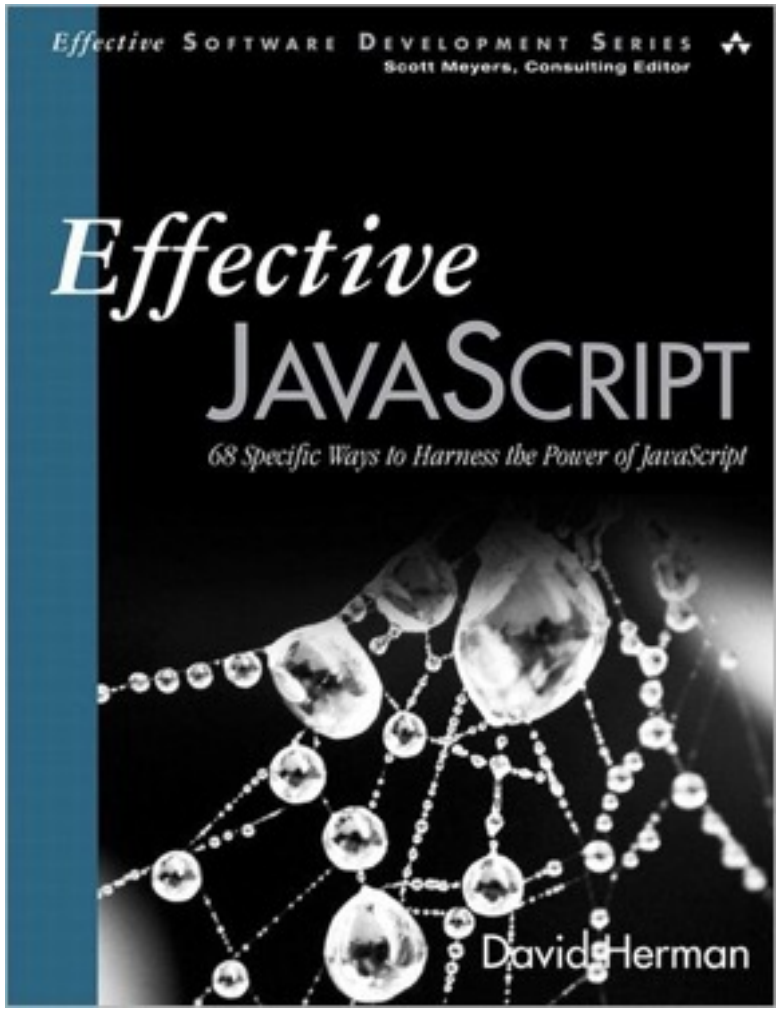


Where to go from here?

- Warmly recommended: Doug Crockford on JavaScript http://goo.gl/FGxmM (YouTube playlist)



Where to go from here?



Further references

- ECMAScript 5:
  - “Changes to JavaScript Part 1: EcmaScript 5” (Mark S. Miller, Waldemar Horwat, Mike Samuel), Google Tech Talk (May 2009)
  - “Secure Mashups in ECMAScript 5” (Mark S. Miller), QCon 2012 Talk http://www.infoq.com/presentations/Secure-Mashups-in-ECMAScript-5
- ES6 latest developments: http://wiki.ecmascript.org and the es-discuss@mozilla.org mailing list.
- ES6 Modules: http://www.2ality.com/2013/07/es6-modules.html
- R. Mark Volkmann: “Using ES6 Today!” http://sett.ociweb.com/sett/settApr2014.html

Thanks for listening!

ECMAScript 5 and 6

The present and future of JavaScript

Tom Van Cutsem

@tvcutsem