The road to ES6, and beyond

A tale about JavaScript's past, present and future

Tom Van Cutsem

jsconf.be 2015

@tvcutsem

My involvement in JavaScript

Vrije Universiteit Brussel

- 2004-2008: built up expertise in programming languages research during my PhD

- 2010: Visiting Faculty at Google, joined Caja team
- Joined ECMA TC39 (Javascript standardization committee)

ecma INTERNATIONAL

- Actively contributed to the ECMAScript 6 specification

Talk Outline

- Part I: JavaScript’s past, and the long road to ECMAScript 6
- Part II: a brief tour of ECMAScript 6
- Part III: using ECMAScript 6 today, and what lies beyond
- Wrap-up

Part I
JavaScript’s past, and the long road to ECMAScript 6

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”

Brendan Eich,
Inventor of JavaScript

Netscape®

The world’s most misunderstood language

Douglas Crockford,
Inventor of JSON

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

Exhuming the terror of JavaScript

JavaScript:
The Evil Parts

```
for (p=0;p<2;p++)
document.write(
 eval("publisher_"+p))
```

Gregor Richards

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

ECMAScript: “Standard” JavaScript

Carakan

Chakra

Spidermonkey

V8

JavaScriptCore

ECMAScript: “Standard” JavaScript

Carakan

Chakra

Spidermonkey

V8

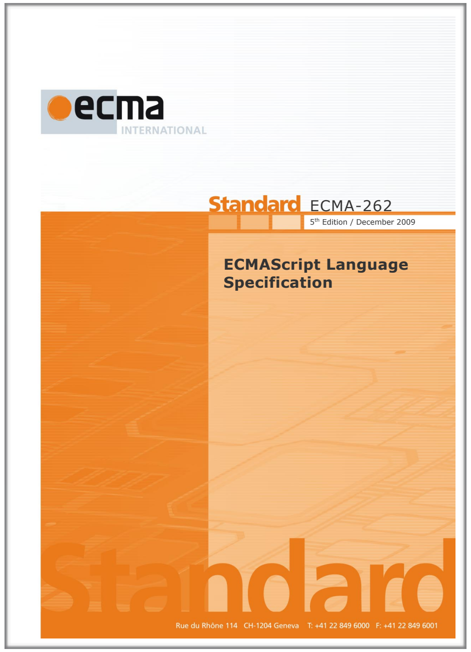
JavaScriptCore

node JS

TC39: the JavaScript “standardisation committee”

- Representatives from major Internet companies, browser vendors, web organisations, popular JS libraries and academia
- Maintains the ECMA-262 specification.
- The spec is a handbook mainly intended for language implementors. Extremely detailed to reduce incompatibilities.

Allen Wirfs-Brock,
ECMA-262 technical editor



ecma INTERNATIONAL

Standard ECMA-262

5th Edition / December 2009

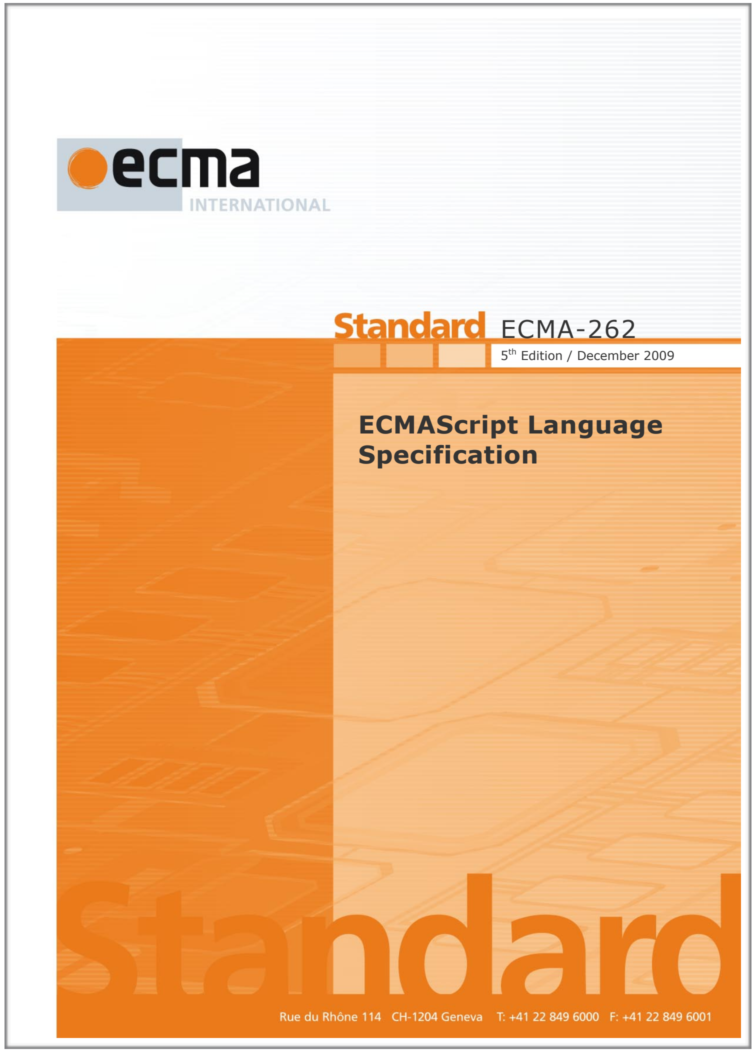
ECMAScript Language Specification

Standard

Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001

ECMAScript specification: history

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



ecma INTERNATIONAL

Standard ECMA-262

5th Edition / December 2009

ECMAScript Language Specification

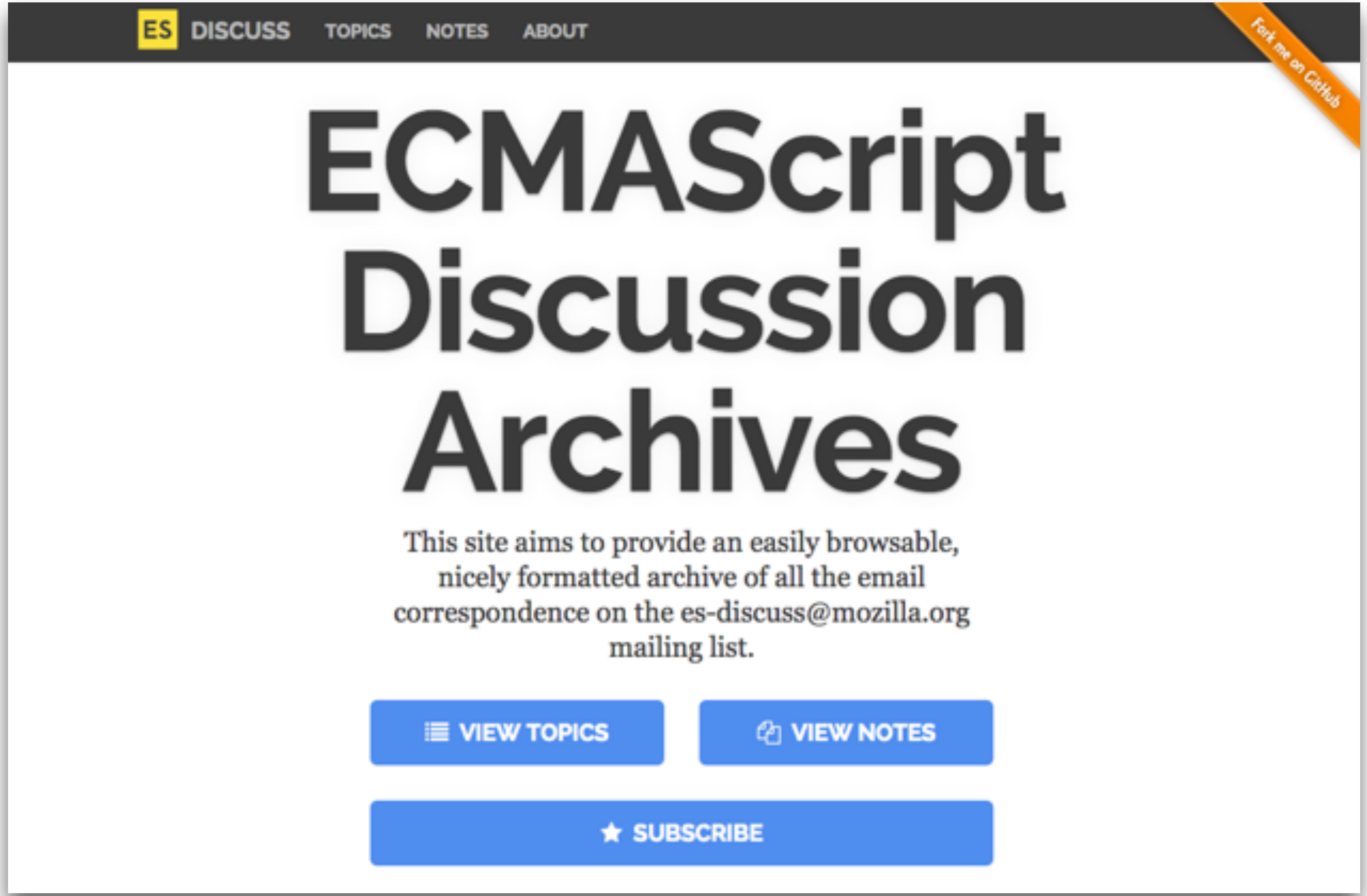
Standard

Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001

TC39

- Meets bi-monthly, mostly in the SF bay area. **Meetings** are technical, not political in nature
- **Discussions** held in the open on es-discuss@mozilla.org
- Committee very much aware of the dangers of “design-by-committee”.
  - **Champion** model to combat this (each feature led by handful of experts)
- Important **decisions** made by global consensus

esdiscuss.org is your friend

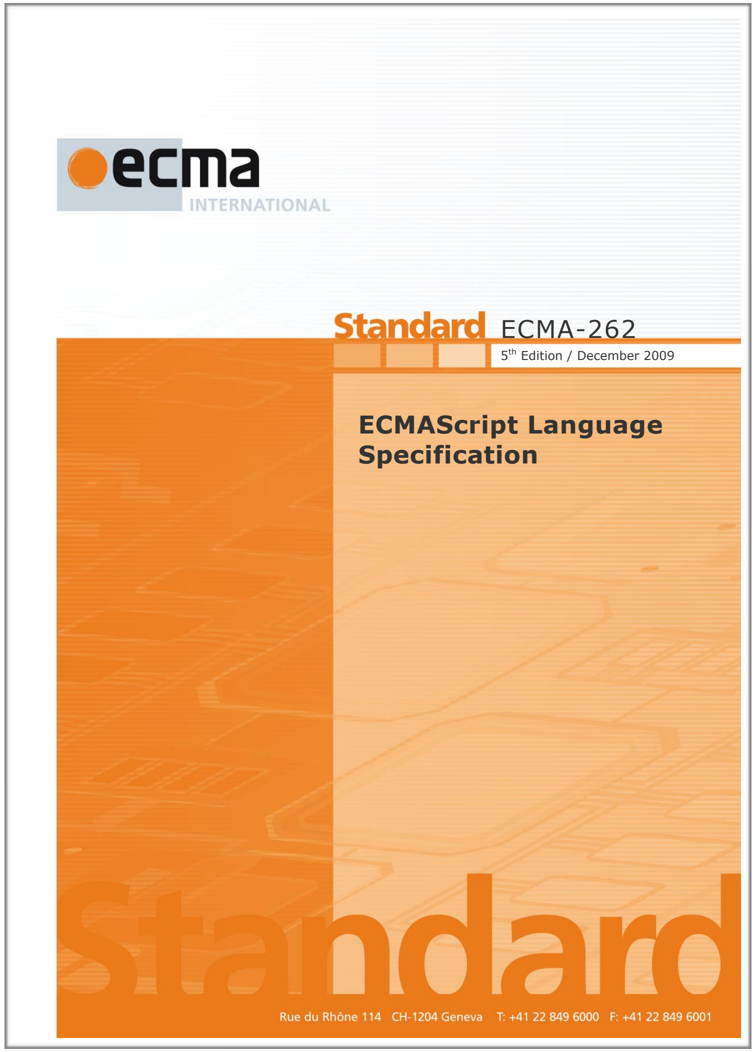


ECMAScript Discussion Archives

This site aims to provide an easily browsable, nicely formatted archive of all the email correspondence on the es-discuss@mozilla.org mailing list.

ECMAScript 5

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



ecma
INTERNATIONAL

Standard ECMA-262
5^{th} Edition / December 2009

ECMAScript Language Specification

Standard

Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001

Ecmascript 5 Strict mode

- How many of you have heard of ECMAScript 5 strict mode?
- How many of you are writing all of their code in strict mode?

Ecmascript 5 Strict mode

- Safer, more robust, subset of the language
- Why? Among others:
  - No silent errors
  - True static scoping rules
- Enabler for the larger ECMAScript 6 effort

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)

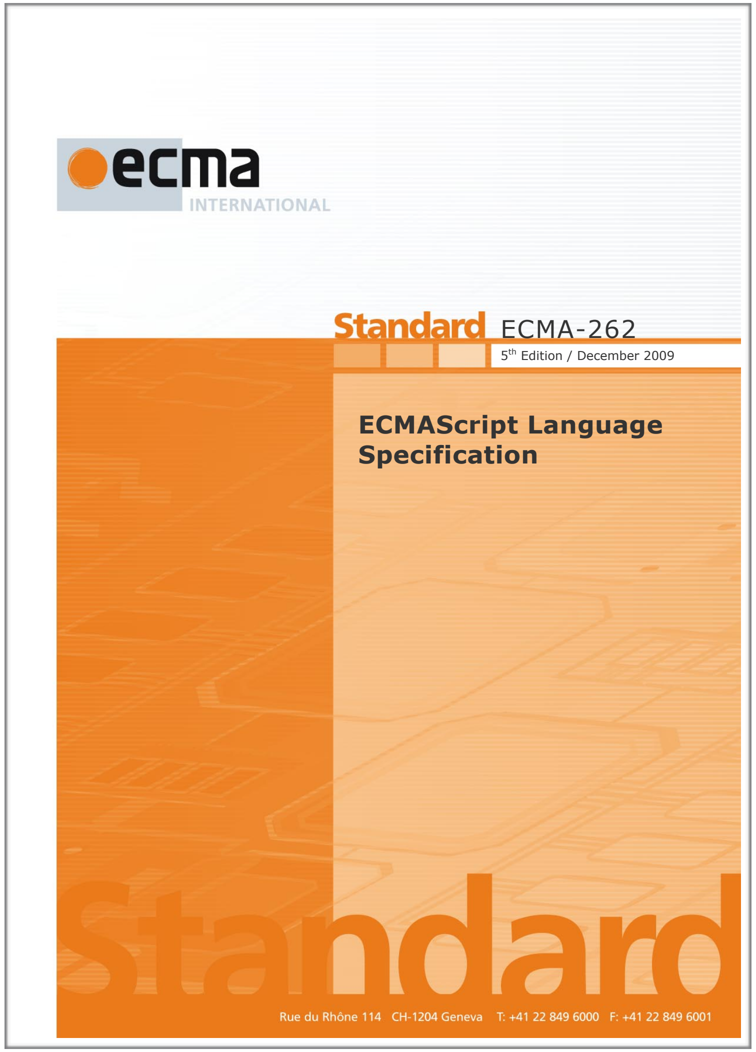
```
function f() {
  "use strict";
  var xfoo;
  xFoo = 1;  // error: assigning to an undeclared variable
}
```

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

Part II
A brief tour of ECMAScript 6

ECMAScript specification

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



ecma
INTERNATIONAL

Standard ECMA-262
5th Edition / December 2009

ECMAScript Language Specification

Standard

Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001

ECMAScript 6

- Major update: many new features (too many to list here)

- Point-in-case:

ES5.1

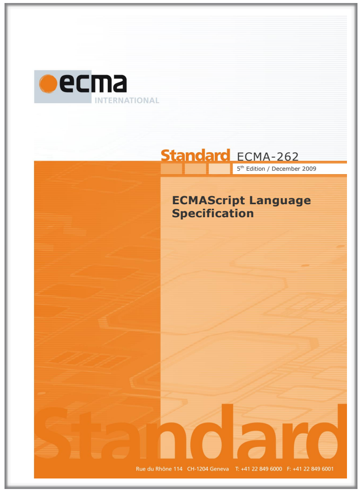
ecma
INTERNATIONAL
Standard ECMA-262
5th Edition / December 2009
ECMAScript Language Specification
Standard

258-page pdf

x 2.4

ES6 draft

rev 37 (april 2015)


ecma
INTERNATIONAL
Standard ECMA-262
5th Edition / December 2009
ECMAScript Language Specification
Standard
Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001

613-page pdf

ECMAScript 6

- Major update: many new features (too many to list here)
- Recommended reading: Luke Hoban's overview at

git.io/es6features

Luke Hoban,
Microsoft representative on TC39

ECMAScript 6

- I will focus on the following loose themes:
  - Improving functions
  - Improving modularity
  - Improving control flow
  - Improving collections
  - Improving reflection (time permitting)

ECMAScript 6: improving functions

- Arrow functions
- Rest arguments
- Optional arguments
- Destructuring
- Improved function scoping: let + const
- Tail calls

ECMAScript 6: arrow functions

- Shorter, and also automatically captures current value of `this`
  No more `var that = 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`
  No more `var that = 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

- By default, body of an arrow function parsed as an *expression*
- If you want to write a *statement*, use curlies:

```
function sumPositive(array) {
  let sum = 0;
  array.forEach(x => {
    if (x > 0) { sum += x; }
  });
  return sum;
}
```

- If you want to return an object, wrap parens around the curlies:

```
angles.map((a) => ({ cos: Math.cos(a), sin: Math.sin(a) }))
```

ECMAScript 6: arrow functions

- By default, body of an arrow function parsed as an *expression*

- If you want to write a *statement*, use curlies:

```
function sumPositive(array) {
  let sum = 0;
  array.forEach(x => {
    if (x > 0) { sum += x; }
  });
  return sum;
}
```

- If you want to return an object, wrap parens around the curlies:

```
angles.map((a) => ({ cos: Math.cos(a), sin: Math.sin(a) }))
```

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: destructuring

- Not just arrays, also for objects:

ES5

```
var node = binaryTree.findNode(key);
var left = (node !== undefined ? node.left : node);
var right = (node !== undefined ? node.right : node);
```

ES6

```
var { left, right } = binaryTree.findNode(key);
```

ECMAScript 6: destructuring

- Not just arrays, also for objects:

ES5

```
var node = binaryTree.findNode(key);
var left = (node !== undefined ? node.left : node);
var right = (node !== undefined ? node.right : node);
```

⇩

ES6

```
var { left, right } = binaryTree.findNode(key);
```

ECMAScript 6: destructuring

- Can do destructuring in parameter position. This gives us elegant keyword parameters!

ES5

```
function fetchRows(options) {
  var args = (options === undefined ? {} : options);
  var limit = (args.limit === undefined ? 10 : args.limit);
  var offset = (args.offset === undefined ? 0 : args.offset);
  var orderBy = (args.orderBy === undefined ? “id” : args.orderBy);
  console.log(limit, offset, orderBy); …
});
```

ES6

```
function fetchRows({ limit=10, offset=0, orderBy=“id”}) {
  console.log(limit, offset, orderBy); …
});
```

ECMAScript 6: destructuring

- Can do destructuring in parameter position. This gives us elegant keyword parameters!

ES5

```
function fetchRows(options) {
  var args = (options === undefined ? {} : options);
  var limit = (args.limit === undefined ? 10 : args.limit);
  var offset = (args.offset === undefined ? 0 : args.offset);
  var orderBy = (args.orderBy === undefined ? “id” : args.orderBy);
  console.log(limit, offset, orderBy); …
});
```

⇩

ES6

```
function fetchRows({ limit=10, offset=0, orderBy=“id”}) {
  console.log(limit, offset, orderBy); …
});
```

ECMAScript 6: let + const

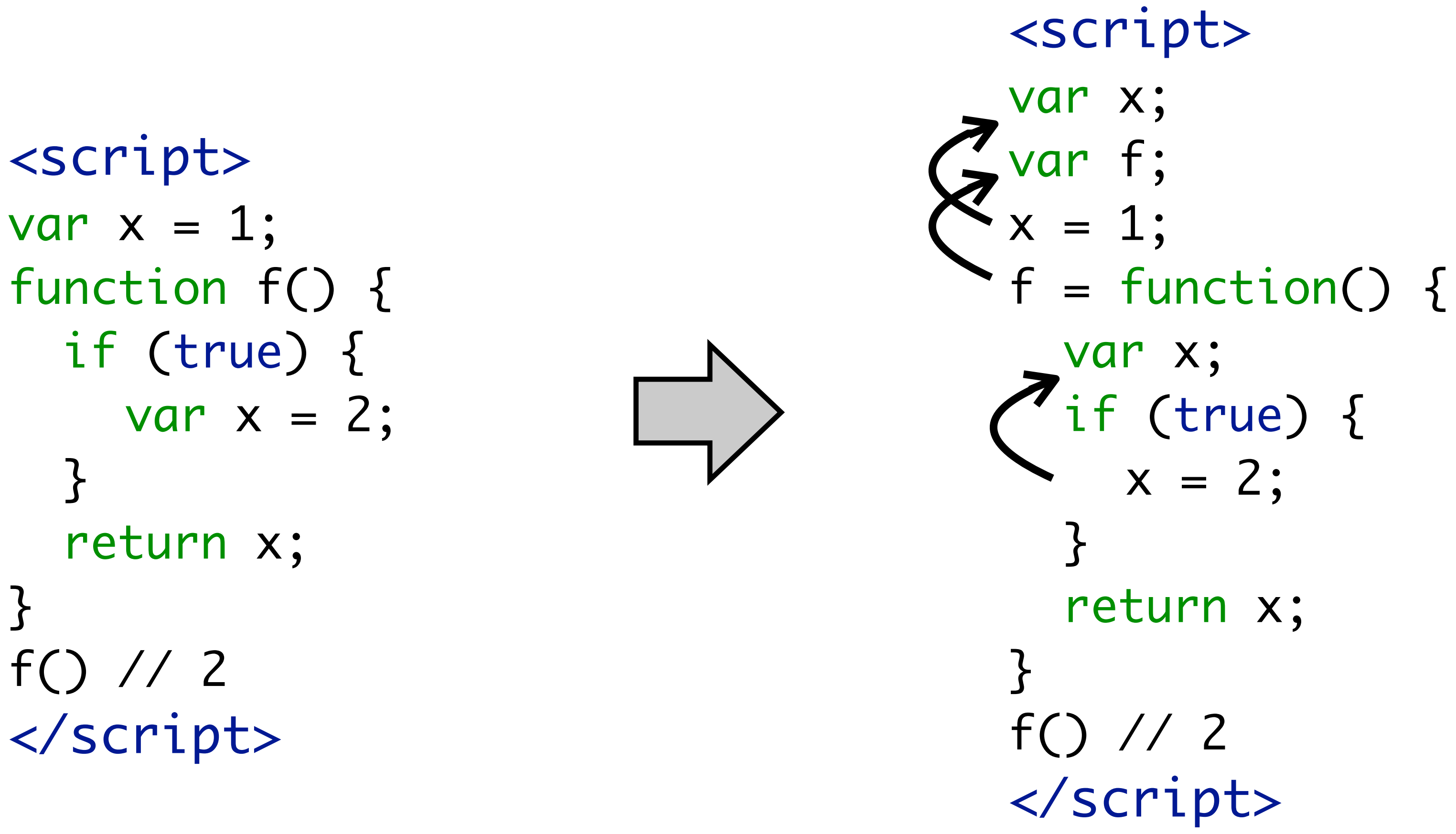
- Remember “var hoisting”?

- JavaScript uses block *syntax*, but does not use block *scope*

```
<script>
var x = 1;
function f() {
  if (true) {
    var x = 2;
  }
  return x;
}
f()
</script>
```

ECMAScript 6: let + const

- Variable declarations are “hoisted” to the beginning of the function body



```
<script>
var x = 1;
function f() {
  if (true) {
    var x = 2;
  }
  return x;
}
f() // 2
</script>
```

```
<script>
var x;
var f;
x = 1;
f = function() {
  var x;
  if (true) {
    x = 2;
  }
  return x;
}
f() // 2
</script>
```

ECMAScript 6: let + const

- Let-declarations are truly block-scoped
- “let is the new var”

ES5

```
<script>
var x = 1;
function f() {
  if (true) {
    var x = 2;
  }
  return x;
}
f() // 2
</script>
```

ES6

```
<script>
let x = 1;
function f() {
  if (true) {
    let x = 2;
  }
  return x;
}
f() // 1
</script>
```

ECMAScript 6: let + const

- Let-declarations are truly block-scoped
- “let is the new var”

ES5

```
<script>
var x = 1;
function f() {
  if (true) {
    var x = 2;
  }
  return x;
}
f() // 2
</script>
```

ES6

```
<script>
let x = 1;
function f() {
  if (true) {
    let x = 2;
  }
  return x;
}
f() // 1
</script>
```

ECMAScript 6: let + const

- Const-declarations are single-assignment
- Static restrictions prevent use before assignment
- More like Java “final” than C++ “const”: the *value* referred to by a const variable may still change

```
function f() {
  const x = g(x);           // static error
  const y = { message: “hello” };
  y = { message: “world” }; // static error
  y.message = “world”;      // ok
}
```

ECMAScript 6: tail calls

- Calls in tail-position guaranteed not to consume stack space
- Makes recursive algorithms practical for large inputs

```
function count(list, acc = 0) {
  if (!list) {
    return acc;
  }
  return count(list.next, acc + 1);
}
```

ES5

```
count(makeList(1000000));
// Error: StackOverflow
```

→

ES6

```
count(makeList(1000000));
// 1000000
```

ECMAScript 6: improving modularity

- Classes (with single-inheritance)
- Enhanced object literals
- 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: classes

- Single-inheritance, super-calls, static members

```
class Point3D extends Point {
  constructor(x, y, z) {
    super(x,y);
    this.z = z;
  }

  static getOrigin() {
    return new Point3D(0,0,0);
  }
}
```

ECMAScript 6: enhanced object literals

- New syntax within object literals in-line with new class syntax

```
var parent = {…};
var foo = 0;
var key = “hello”;
var obj = {
  foo: foo,
  toString: function() {
    return “foo”;
  }
};
obj.__proto__ = parent;
obj[key] = 42;
```

```
var parent = {…};
var foo = 0;
var key = “hello”;
var obj = {
  __proto__: parent,
  foo,
  toString() {
    return “foo”;
  },
  [key]: 42
};
```

ECMAScript 6: enhanced object literals

- New syntax within object literals in-line with new class syntax

```
var parent = {…};
var foo = 0;
var key = “hello”;
var obj = {
  foo: foo,
  toString: function() {
    return “foo”;
  }
};
obj.__proto__ = parent;
obj[key] = 42;
```

⇨

```
var parent = {…};
var foo = 0;
var key = “hello”;
var obj = {
  __proto__: parent,
  foo,
  toString() {
    return “foo”;
  },
  [key]: 42
};
```

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: modules

- There is much more to be said about modules
- Module loader API
  - Dynamic (async) module loading
  - Compilation hooks (e.g. transform cs to js at load-time)
  - Load code in isolated environments with their own global object
- Inspiration from popular JS module systems like commonjs, requireJS

ECMAScript 6: improving control flow

- Iterators
- Generators
- Promises
- async/await [tentative ES7 sneak peek]

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 }
    }
  }
}
```

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 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 template strings

- String interpolation (e.g. for templating) is very common in JS
- Vulnerable to injection attacks

```
function createDiv(input) {
  return "<div>"+input+"</div>";
};

createDiv("</div><script>…");
// "<div></div><script>…</div>"
```

ECMAScript 6 template strings

- Template strings combine convenient syntax for interpolation with a way of automatically building the string

```
function createDiv(input) {
  return html`<div>${input}</div>`;
};

createDiv("</div><script>…");
// "<div>&lt;/div&gt;&lt;script&gt;…</div>"
```

ECMAScript 6 template strings

- User-extensible: just sugar for a call to a template function
- Expectation that browser will provide html, css template functions

```
function createDiv(input) {
  return html(["<div>","</div>"], input);
};

createDiv("</div><script>…");
// "<div>&lt;/div&gt;&lt;script&gt;…</div>"
```

ECMAScript 6 template strings

- The template tag is optional. If omitted, just builds a string.

```
let str = `1 plus 2 is ${1 + 2}`;
```

- And yes, template strings can span multiple lines, so we finally have multi-line strings:

```
function createPoem() {
  return `hello
          world`;
};
```

ECMAScript 6 template strings: closing note

- Template strings are not to be confused with template languages such as handlebars, mustache, etc.
    - Often used to generate strings
    - Contain instructions such as loops, conditionals, etc.

ECMAScript 6: improving collections

- Up to ES5: arrays and objects. Objects (ab?)used as maps of String to Any

- ES6 brings Map, Set, WeakMap, WeakSet

```
let m = new Map();
m.set(“a”, 42);
m.get(“a”) === 42;
```

- Also support Objects as keys (not just Strings)

- Weak* variants automatically remove entry when key becomes garbage. Ideal for building caches.

ECMAScript 6: improving reflection

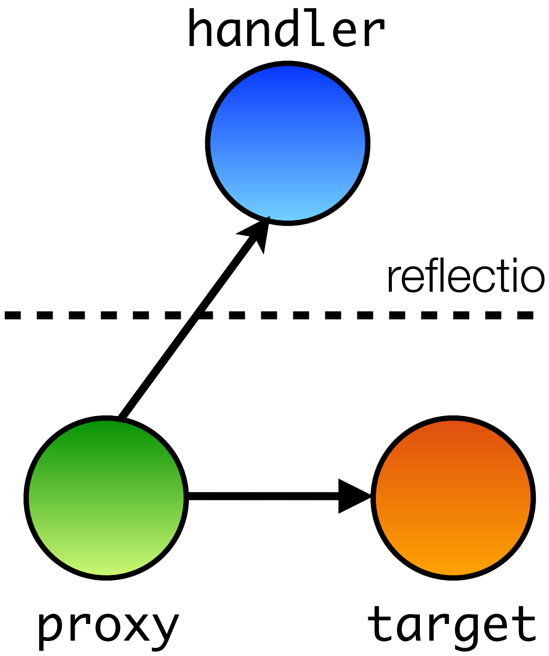
- Proxies
  - Dynamic proxy objects: objects whose behavior can be controlled in JavaScript itself
  - Useful to create *generic* (i.e. type-independent) object wrappers

ECMAScript 6 proxies

```
var proxy = new Proxy(target, handler);
```

```
handler.get(target, ‘foo’)

handler.set(target, ‘foo’, 42)
```



reflection

application

```
proxy.foo

proxy.foo = 42
```

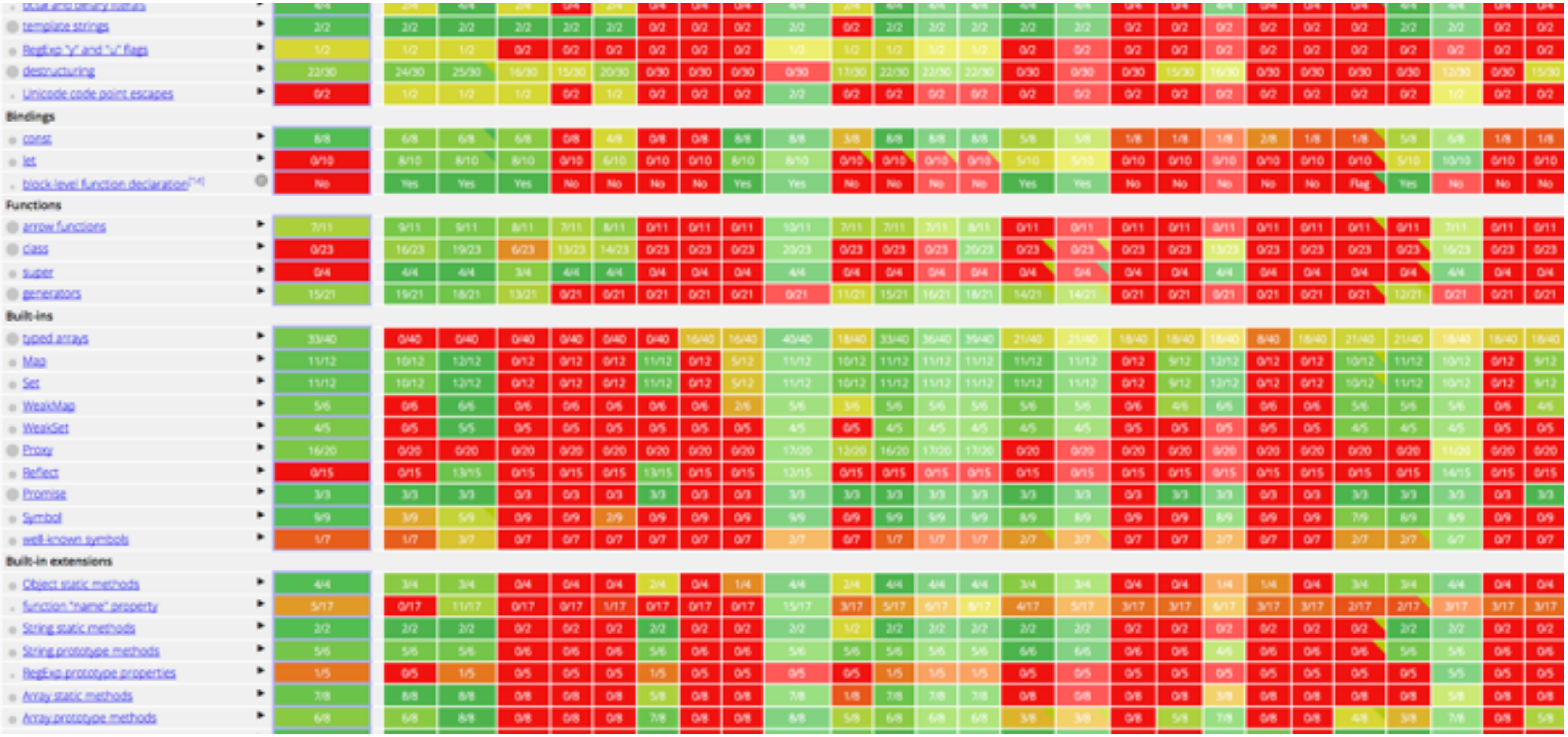
Part III
Using ECMAScript 6 today, and what lies beyond

ECMAScript 6: timeline

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

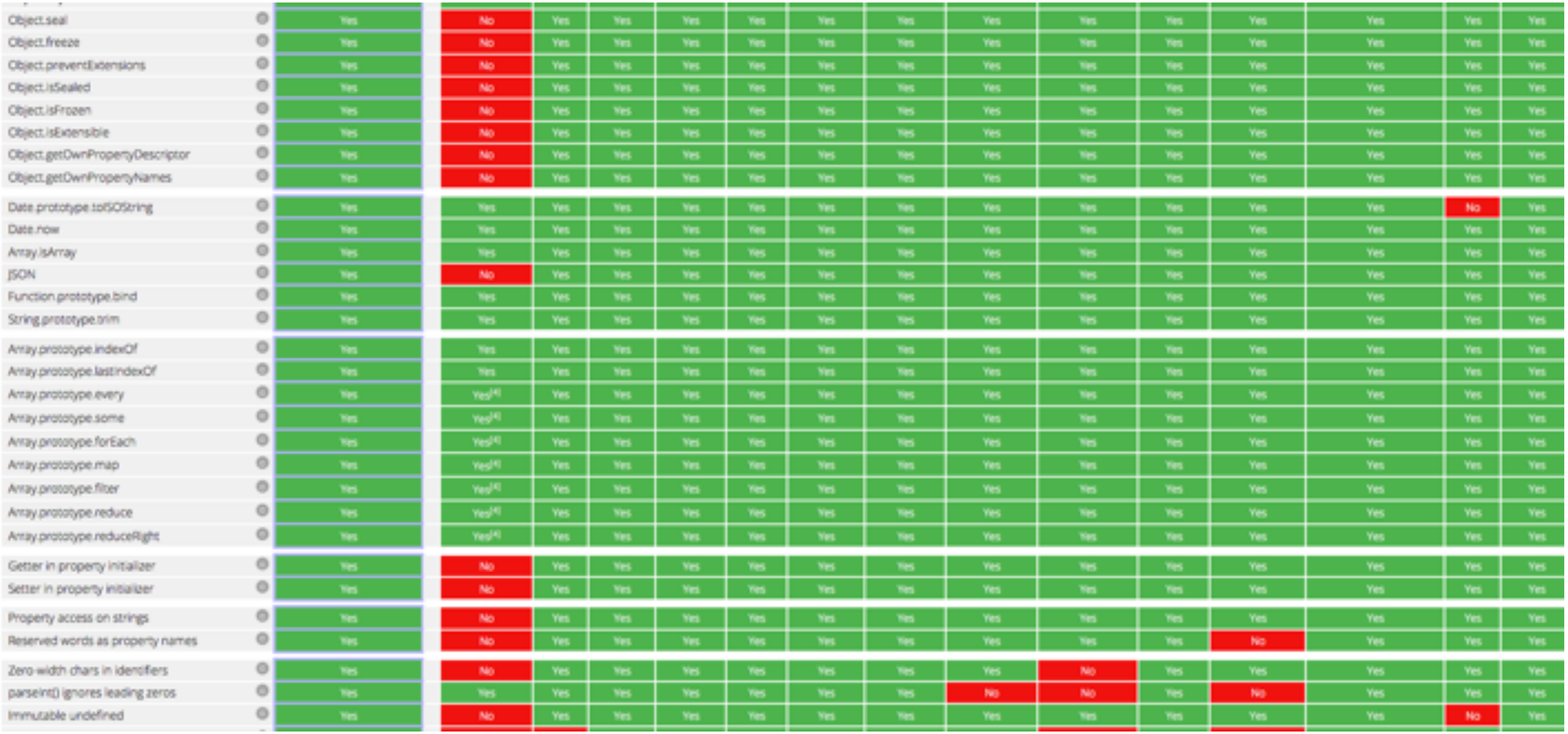
* see Juriy Zaytsev’s (a.k.a. kangax) excellent compatibility tables http://kangax.github.io/es5-compat-table/es6/ for current status

ECMAScript 6 support (april 2015)



octal and binary literals	4/4	2/4	4/4	2/4	0/4	2/4	0/4	0/4	0/4	4/4	2/4	4/4	4/4	4/4	4/4	4/4	0/4	0/4	4/4	0/4	0/4	0/4	4/4	4/4	0/4	0/4
template strings	2/2	2/2	2/2	2/2	2/2	2/2	0/2	0/2	0/2	2/2	0/2	2/2	2/2	2/2	2/2	2/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	0/2	0/2
RegExp "y" and "u" flags	1/2	1/2	1/2	0/2	0/2	0/2	0/2	0/2	0/2	1/2	1/2	1/2	1/2	1/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2
destructuring	22/30	24/30	25/30	16/30	15/30	20/30	0/30	0/30	0/30	0/30	17/30	22/30	22/30	22/30	0/30	0/30	0/30	15/30	16/30	0/30	0/30	0/30	0/30	12/30	0/30	15/30
Unicode code point escapes	0/2	1/2	1/2	1/2	0/2	1/2	0/2	0/2	0/2	2/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	1/2	0/2	0/2
Bindings																										
const	8/8	6/8	6/8	6/8	0/8	4/8	0/8	0/8	8/8	8/8	3/8	8/8	8/8	8/8	5/8	5/8	1/8	1/8	1/8	2/8	1/8	1/8	5/8	6/8	1/8	1/8
let	0/10	8/10	8/10	8/10	0/10	6/10	0/10	0/10	8/10	8/10	0/10	0/10	0/10	0/10	5/10	5/10	0/10	0/10	0/10	0/10	0/10	0/10	5/10	10/10	0/10	0/10
block-level function declaration[14]	No	Yes	Yes	Yes	No	No	No	No	Yes	Yes	No	No	No	No	Yes	Yes	No	No	No	No	No	Flag	Yes	No	No	No
Functions																										
arrow functions	7/11	9/11	9/11	8/11	7/11	8/11	0/11	0/11	0/11	10/11	7/11	7/11	7/11	8/11	0/11	0/11	0/11	0/11	0/11	0/11	0/11	0/11	0/11	7/11	0/11	0/11
class	0/23	16/23	19/23	6/23	13/23	14/23	0/23	0/23	0/23	20/23	0/23	0/23	0/23	20/23	0/23	0/23	0/23	0/23	13/23	0/23	0/23	0/23	0/23	16/23	0/23	0/23
super	0/4	4/4	4/4	3/4	4/4	4/4	0/4	0/4	0/4	4/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	4/4	0/4	0/4	0/4	0/4	4/4	0/4	0/4
generators	15/21	19/21	18/21	13/21	0/21	0/21	0/21	0/21	0/21	0/21	11/21	15/21	16/21	18/21	14/21	14/21	0/21	0/21	0/21	0/21	0/21	0/21	12/21	0/21	0/21	0/21
Built-ins																										
typed arrays	33/40	0/40	0/40	0/40	0/40	0/40	0/40	16/40	16/40	40/40	18/40	33/40	36/40	39/40	21/40	21/40	18/40	18/40	18/40	8/40	18/40	21/40	21/40	18/40	18/40	18/40
Map	11/12	10/12	12/12	0/12	0/12	0/12	11/12	0/12	5/12	11/12	10/12	11/12	11/12	11/12	11/12	11/12	0/12	9/12	12/12	0/12	0/12	10/12	11/12	10/12	0/12	9/12
Set	11/12	10/12	12/12	0/12	0/12	0/12	11/12	0/12	5/12	11/12	10/12	11/12	11/12	11/12	11/12	11/12	0/12	9/12	12/12	0/12	0/12	10/12	11/12	10/12	0/12	9/12
WeakMap	5/6	0/6	6/6	0/6	0/6	0/6	0/6	0/6	2/6	5/6	3/6	5/6	5/6	5/6	5/6	5/6	0/6	4/6	6/6	0/6	0/6	5/6	5/6	5/6	0/6	4/6
WeakSet	4/5	0/5	5/5	0/5	0/5	0/5	0/5	0/5	0/5	4/5	0/5	4/5	4/5	4/5	4/5	4/5	0/5	0/5	0/5	0/5	0/5	4/5	4/5	4/5	0/5	0/5
Proxy	16/20	0/20	0/20	0/20	0/20	0/20	0/20	0/20	0/20	17/20	12/20	16/20	17/20	17/20	0/20	0/20	0/20	0/20	0/20	0/20	0/20	0/20	0/20	11/20	0/20	0/20
Reflect	0/15	0/15	13/15	0/15	0/15	0/15	13/15	0/15	0/15	12/15	0/15	0/15	0/15	0/15	0/15	0/15	0/15	0/15	0/15	0/15	0/15	0/15	0/15	14/15	0/15	0/15
Promise	3/3	3/3	3/3	0/3	0/3	0/3	3/3	0/3	0/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	0/3	3/3	3/3	0/3	0/3	3/3	3/3	3/3	0/3	3/3
Symbol	9/9	3/9	5/9	0/9	0/9	2/9	0/9	0/9	0/9	9/9	0/9	9/9	9/9	9/9	8/9	8/9	0/9	0/9	8/9	0/9	0/9	7/9	8/9	8/9	0/9	0/9
well-known symbols	1/7	1/7	3/7	0/7	0/7	0/7	0/7	0/7	0/7	2/7	0/7	1/7	1/7	1/7	2/7	2/7	0/7	0/7	2/7	0/7	0/7	2/7	2/7	6/7	0/7	0/7
Built-in extensions																										
Object static methods	4/4	3/4	3/4	0/4	0/4	0/4	2/4	0/4	1/4	4/4	2/4	4/4	4/4	4/4	3/4	3/4	0/4	0/4	1/4	1/4	0/4	3/4	3/4	4/4	0/4	0/4
function "name" property	5/17	0/17	11/17	0/17	0/17	1/17	0/17	0/17	0/17	15/17	3/17	5/17	6/17	8/17	4/17	5/17	3/17	3/17	6/17	3/17	3/17	2/17	2/17	3/17	3/17	3/17
String static methods	2/2	2/2	2/2	0/2	0/2	0/2	2/2	0/2	0/2	2/2	1/2	2/2	2/2	2/2	2/2	2/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	0/2	0/2
String.prototype methods	5/6	5/6	5/6	0/6	0/6	0/6	5/6	0/6	0/6	5/6	5/6	5/6	5/6	5/6	6/6	6/6	0/6	0/6	4/6	0/6	0/6	0/6	5/6	5/6	0/6	0/6
RegExp.prototype properties	1/5	0/5	1/5	0/5	0/5	0/5	1/5	0/5	0/5	0/5	0/5	1/5	1/5	1/5	0/5	0/5	0/5	0/5	0/5	0/5	0/5	0/5	0/5	5/5	0/5	0/5
Array static methods	7/8	8/8	8/8	0/8	0/8	0/8	5/8	0/8	0/8	7/8	1/8	7/8	7/8	7/8	0/8	0/8	0/8	0/8	3/8	0/8	0/8	0/8	0/8	5/8	0/8	0/8
Array.prototype methods	6/8	6/8	8/8	0/8	0/8	0/8	7/8	0/8	0/8	8/8	5/8	6/8	6/8	6/8	3/8	3/8	0/8	5/8	7/8	0/8	0/8	4/8	3/8	7/8	0/8	5/8
]

ECMAScript 5 support (april 2015)



Object.seal	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.freeze	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.preventExtensions	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.isSealed	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.isFrozen	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.isExtensible	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.getOwnPropertyDescriptor	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.getOwnPropertyNames	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Date.prototype.toISOString	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
Date.now	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.isArray	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
JSON	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Function.prototype.bind	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
String.prototype.trim	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.indexOf	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.lastIndexOf	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.every	Yes	Yes[4]	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.some	Yes	Yes[4]	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.forEach	Yes	Yes[4]	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.map	Yes	Yes[4]	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.filter	Yes	Yes[4]	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.reduce	Yes	Yes[4]	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.reduceRight	Yes	Yes[4]	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Getter in property initializer	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Setter in property initializer	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Property access on strings	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Reserved words as property names	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
Zero-width chars in identifiers	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
parseInt() ignores leading zeros	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	No	Yes	Yes	Yes
Immutable undefined	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes

ECMAScript 6 compilers

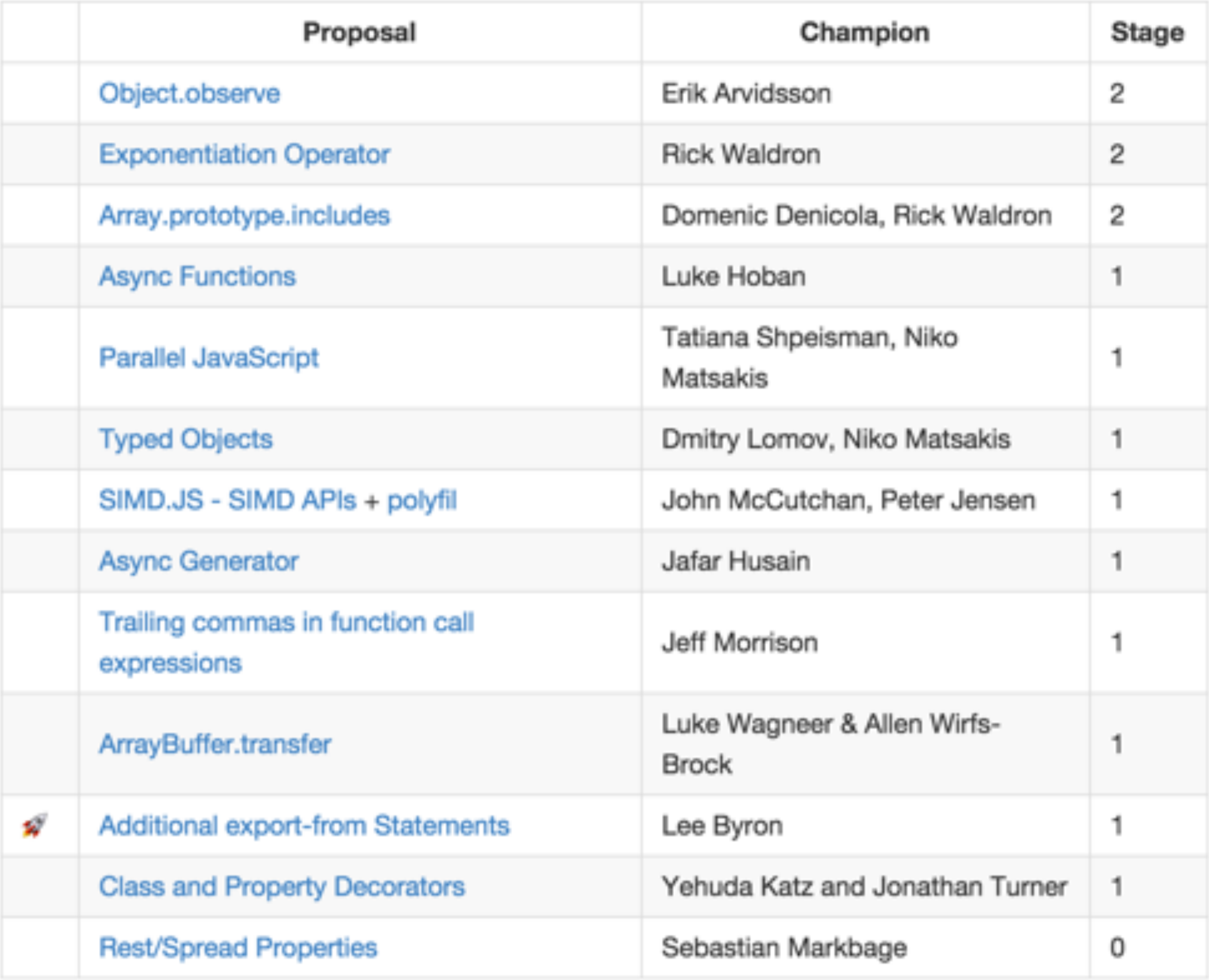
- Compile ECMAScript 6 to ECMAScript 5
- Google **Traceur**: mature and quite feature-complete. Aims to be fully spec-compliant.
- **Babel**: focus on producing readable (as-if hand-written) ES5 code. Supports JSX.
- Microsoft **TypeScript**: technically not ES6 but roughly a superset of ES6. Bonus: type inference and optional static typing.

Going forward

- ECMAScript 6 officially called “ECMAScript 2015”
- Goal is to have yearly spec releases from now on
- Hence, not sure there will ever be an “ECMAScript 7” as such

ES7 Proposals on the table

- Again, too many to list in detail. See https://github.com/tc39/ecma262



	Proposal	Champion	Stage
	Object.observe	Erik Arvidsson	2
	Exponentiation Operator	Rick Waldron	2
	Array.prototype.includes	Domenic Denicola, Rick Waldron	2
	Async Functions	Luke Hoban	1
	Parallel JavaScript	Tatiana Shpeisman, Niko Matsakis	1
	Typed Objects	Dmitry Lomov, Niko Matsakis	1
	SIMD.JS - SIMD APIs + polyfil	John McCutchan, Peter Jensen	1
	Async Generator	Jafar Husain	1
	Trailing commas in function call expressions	Jeff Morrison	1
	ArrayBuffer.transfer	Luke Wagneer & Allen Wirfs-Brock	1
🚀	Additional export-from Statements	Lee Byron	1
	Class and Property Decorators	Yehuda Katz and Jonathan Turner	1
	Rest/Spread Properties	Sebastian Markbage	0

Wrap-up

Take-home messages

- ECMAScript 5 strict mode: a saner basis for the future evolution of JavaScript

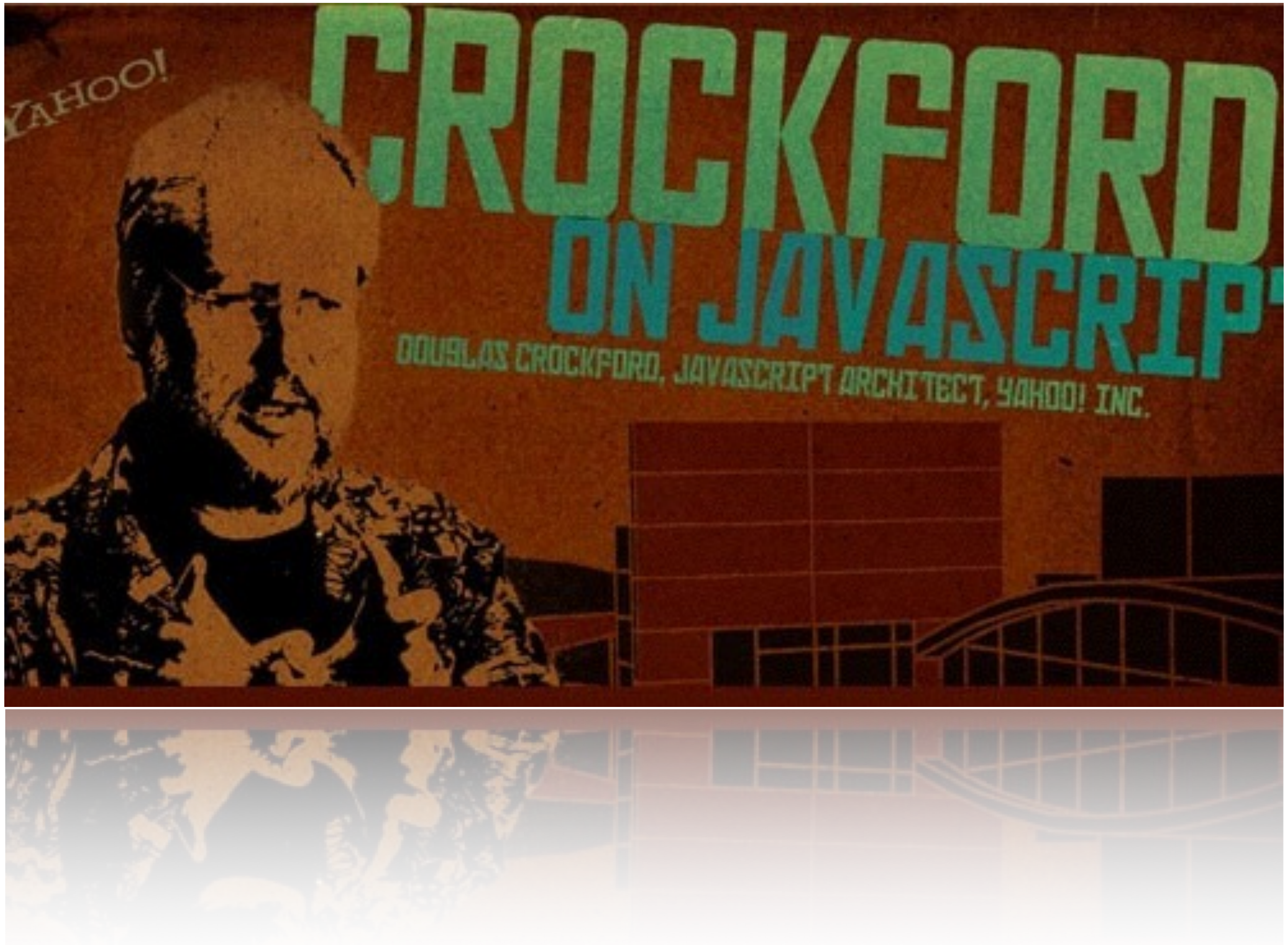
- Opt-in subset that removes some of JavaScript’s warts. Use it!

Take-home messages

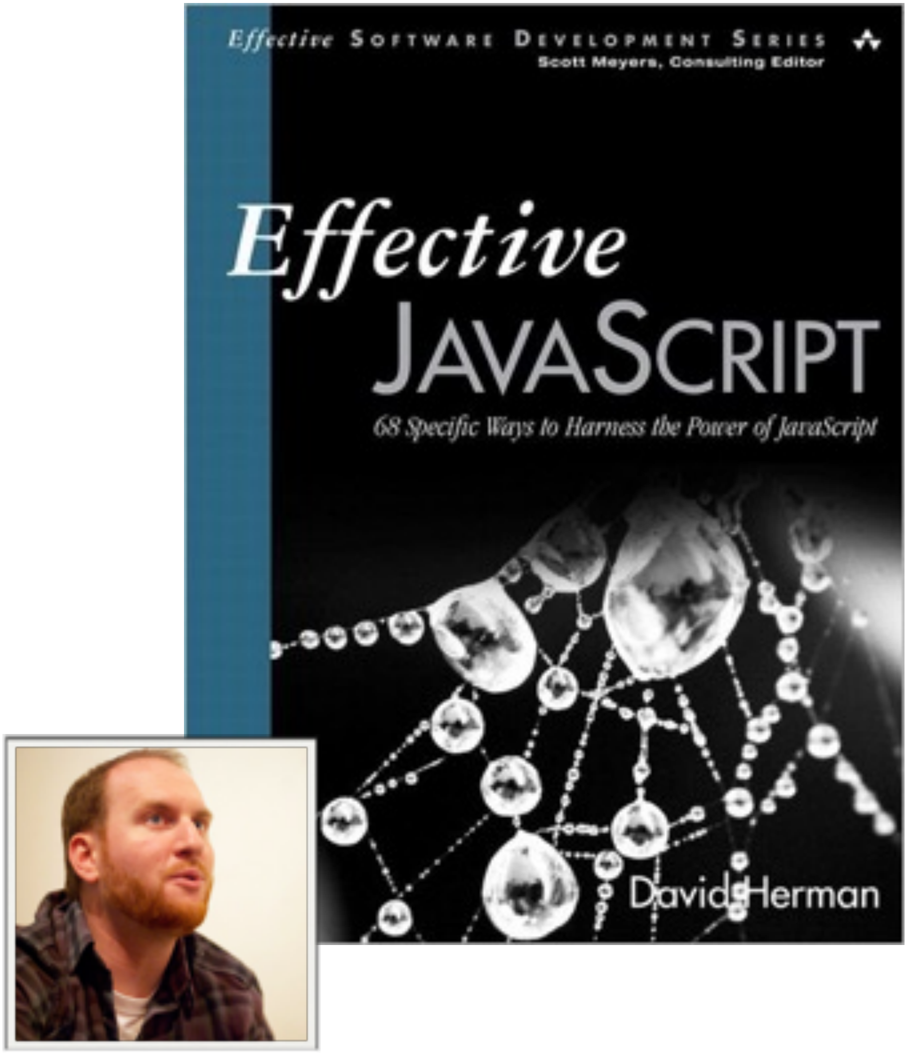
- ECMAScript 6 is a *major* upgrade to the language
- Expect browsers to implement the upgrade gradually and piecemeal
- Use ES6 to ES5 compilers to bridge the gap
- You can use ES6 today!

Where to go from here?

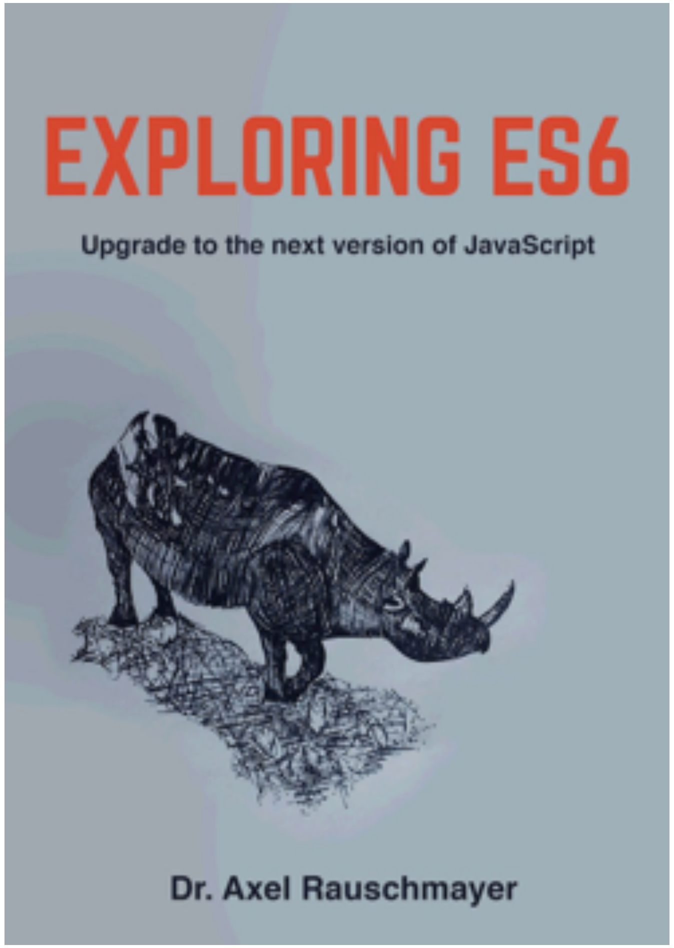
- Warmly recommended: Doug Crockford on JavaScript
  http://goo.gl/FGxmM (YouTube playlist)



Where to go from here?


Effective Software Development Series
Scott Meyers, Consulting Editor
Effective
JavaScript
68 Specific Ways to Harness the Power of JavaScript
David Herman

Dave Herman
Mozilla representative on TC39



Additional references

- ECMAScript 5 and strict mode: “Changes to JavaScript Part 1: EcmaScript 5” (Mark S. Miller, Waldemar Horwat, Mike Samuel), Google Tech Talk (May 2009)

- ECMAScript latest developments: http://wiki.ecmascript.org and the es-discuss@mozilla.org mailing list.

- ECMAScript 6: Axel Rauschmayer’s blog: http://www.2ality.com

- Using ES6 today: R. Mark Volkmann: “Using ES6 Today!” http://sett.ociweb.com/sett/settApr2014.html

Thanks for listening!

The road to ES6, and beyond

A tale about JavaScript’s past, present and future

Tom Van Cutsem

jsconf.be 2015

@tvcutsem