The road to ES6, and beyond
A tale about JavaScript’s past, present and future

Tom Van Cutsem
jsconf.be 2015
My involvement in JavaScript

• 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)

• 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*
The world’s most misunderstood language

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
- ...

JavaScript: The Evil Parts

Exhuming the terror of JavaScript

Gregor Richards
ECMAScript: “Standard” JavaScript

- Chakra
- Carakan
- Spidermonkey
- V8
- JavaScriptCore
ECMAScript: “Standard” JavaScript
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
ECMAScript specification: history

- 1st ed. 1997
- 2nd ed. 1998
- 3rd ed. 1999
- 4th ed.
- 5th ed. 2009
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](mailto: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.

- VIEW TOPICS
- VIEW NOTES
- SUBSCRIBE
ECMAScript 5

- 1st ed. 1997
- 2nd ed. 1998
- 3rd ed. 1999
- 4th ed.
- 5th ed. 2009
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)
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:
    ```javascript
    function f() { var xfoo; xFoo = 1; }
    ```
The following are forbidden in strict mode (signaled as syntax errors):

```javascript
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)

```javascript
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
ECMAScript 6

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

- Point-in-case:

  ES5.1

  258-page pdf

ES6 draft
rev 37 (april 2015)

  x 2.4

  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;`

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

**ES5**

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

**ES6**

```javascript
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

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

ES6

```javascript
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 curly braces:

```javascript
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 curly braces:

```javascript
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:

```javascript
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:

```javascript
angles.map((a) => ({ cos: Math.cos(a), sin: Math.sin(a) }))
```
ECMAScript 6: rest arguments

ES5

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

ES6

```javascript
function printf(format, ...rest) {
    ...
}
```
ECMAScript 6: rest arguments

ES5

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

ES6

```javascript
function printf(format, ...rest) {
    ...
}
```
ECMAScript 6: optional arguments

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

```
function greet(name = "world") {
    return "Hello, " + name;
}
```
ECMAScript 6: optional arguments

ES5

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

ES6

```javascript
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];
}

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

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];
}

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

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

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

ES6

```javascript
var { left, right } = binaryTree.findNode(key);
```
ECMAScript 6: destructuring

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

```javascript
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); …
}
```

ES5

```javascript
function fetchRows() {
    console.log(limit, offset, orderBy); …
}
```

ES6

```javascript
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!

```javascript
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); …
}
```

ES5

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

ES6
ECMAScript 6: let + const

• Remember “var hoisting”?

• JavaScript uses block syntax, but does not use block scope

```javascript
<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

```javascript
<script>
var x = 1;
function f() {
    if (true) {
        var x = 2;
    }
    return x;
}
f() // 2
</script>
```
ECMAScript 6: let + const

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

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

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

```javascript
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

```javascript
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!

```javascript
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();
```

```javascript
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!

```javascript
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();
```

```javascript
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

```javascript
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

```javascript
var parent = {...};
var foo = 0;
var key = “hello”;
var obj = {
    foo: foo,
    toString: function() {
        return “foo”;
    },
};
obj.__proto__ = parent;
obj[key] = 42;
```
ECMAScript 6: enhanced object literals

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

```javascript
var parent = {...};
var foo = 0;
var key = "hello";
var obj = {
    foo: foo,
    toString: function() {
        return "foo";
    }
};
obj.__proto__ = parent;
obj[key] = 42;
```
ECMAScript 6: modules

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

```html
<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!

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

<script type="module">
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

```javascript
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();
}
```

```javascript
// generates 1, 1, 2, 3, 5, 8, 13, 21, ...
```

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

// generates 1, 1, 2, 3, 5, 8, 13, 21, ...
ECMAScript 6 Iterators

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

**ES5**

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

**ES6**

```javascript
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**

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

**ES6**

```javascript
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**

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

**ES6**

```javascript
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

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

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

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

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

ES6

```javascript
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

```javascript
// 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

```javascript
// 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

```javascript
(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

```javascript
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

```javascript
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

```javascript
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

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

createDiv("</div><script>...");
// "<div></div><script>...</"
 ECMAScript 6 template strings

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

```javascript
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

```javascript
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.

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

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

```javascript
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

```javascript
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

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

handler.get(target, 'foo')

handler.set(target, 'foo', 42)
```

application

proxy.foo

```javascript
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
ECMAScript 6 support (April 2015)
ECMAScript 5 support (April 2015)
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

<table>
<thead>
<tr>
<th>Proposal</th>
<th>Champion</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object.observe</td>
<td>Erik Arvidsson</td>
<td>2</td>
</tr>
<tr>
<td>Exponentiation Operator</td>
<td>Rick Waldron</td>
<td>2</td>
</tr>
<tr>
<td>Array.prototype.includes</td>
<td>Domenic Denicola, Rick Waldron</td>
<td>2</td>
</tr>
<tr>
<td>Async Functions</td>
<td>Luke Hoban</td>
<td>1</td>
</tr>
<tr>
<td>Parallel JavaScript</td>
<td>Tatiana Shpeisman, Niko Matsakis</td>
<td>1</td>
</tr>
<tr>
<td>Typed Objects</td>
<td>Dmitry Lomov, Niko Matsakis</td>
<td>1</td>
</tr>
<tr>
<td>SIMD.JS - SIMD APIs + polyfil</td>
<td>John McCutchan, Peter Jensen</td>
<td>1</td>
</tr>
<tr>
<td>Async Generator</td>
<td>Jafar Husain</td>
<td>1</td>
</tr>
<tr>
<td>Trailing commas in function call expressions</td>
<td>Jeff Morrison</td>
<td>1</td>
</tr>
<tr>
<td>ArrayBuffer.transfer</td>
<td>Luke Wagneer &amp; Allen Wirfs-Brock</td>
<td>1</td>
</tr>
<tr>
<td>🔥 Additional export-from Statements</td>
<td>Lee Byron</td>
<td>1</td>
</tr>
<tr>
<td>Class and Property Decorators</td>
<td>Yehuda Katz and Jonathan Turner</td>
<td>1</td>
</tr>
<tr>
<td>Rest/Spread Properties</td>
<td>Sebastian Markbage</td>
<td>0</td>
</tr>
</tbody>
</table>
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](http://goo.gl/FGxmM) (YouTube playlist)
Where to go from here?

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

The road to ES6, and beyond
A tale about JavaScript’s past, present and future

Tom Van Cutsem
jsconf.be 2015