JavaScript: the Good, the Bad, the Strict and the New Parts

Tom Van Cutsem

@tvcutsem

My involvement in JavaScript

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

- Actively contributed to the ECMAScript 6 specification

Talk Outline

- Part I: JavaScript, the **Good** Parts: what features to use
- Part II: JavaScript, the **Bad** Parts: what features to avoid
- Part III: ECMAScript 5 and the **Strict** Parts: standards & strict mode
- Part IV: ECMAScript 6, the **New** Parts: JavaScript's future

Part I: Javascript, the **Good** Parts

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

Good Parts: Functions

- Functions are first-class, may capture lexical variables (closures)

```
var add = function(a,b) {
  return a+b;
}

add(2,3); // 5
```

```
function accumulator(s) {
  return function(n) {
    return s += n;
  }
}

var a = accumulator(0);
a(1); // 1
a(2); // 3
```

```
button.addEventListener('click', function (event) { … });

[1,2,3].map(function (x) { return x + 1; }); // [2,3,4]
```

Good Parts: Objects

- No class declaration needed, literal syntax, arbitrary nesting

```
var bob = {
  name: “Bob”,
  dob: {
    day: 15,
    month: 03,
    year: 1980
  },
  address: {
    street: “Main St.”,
    number: 5,
    zip: 94040,
    country: “USA”
  }
};
```

Good Parts: combining objects and functions

- Functions can act as object constructors and methods

```
function makePoint(i,j) {
  return {
    x: i,
    y: j,
    toString: function() {
      return ‘(’+ this.x +‘,’+ this.y +‘)’;
    }
  };
}

var p = makePoint(2,3);
var x = p.x;
var s = p.toString();
```

A dynamic language...

```
// computed property access and assignment
p.x                     p["x"]
p.x = 42;               p["x"] = 42;

// dynamic method invocation
p.toString();           p["toString"].apply(p, [ ]);

// add new properties to an object at runtime
p.z = 0;

// delete properties from an object at runtime
delete p.x;
```

Delving Deeper

- Some finer points about:
  - Functions as objects
  - Functions as object constructors
  - Inheritance between objects
  - Functions versus methods

Functions

- Functions are objects

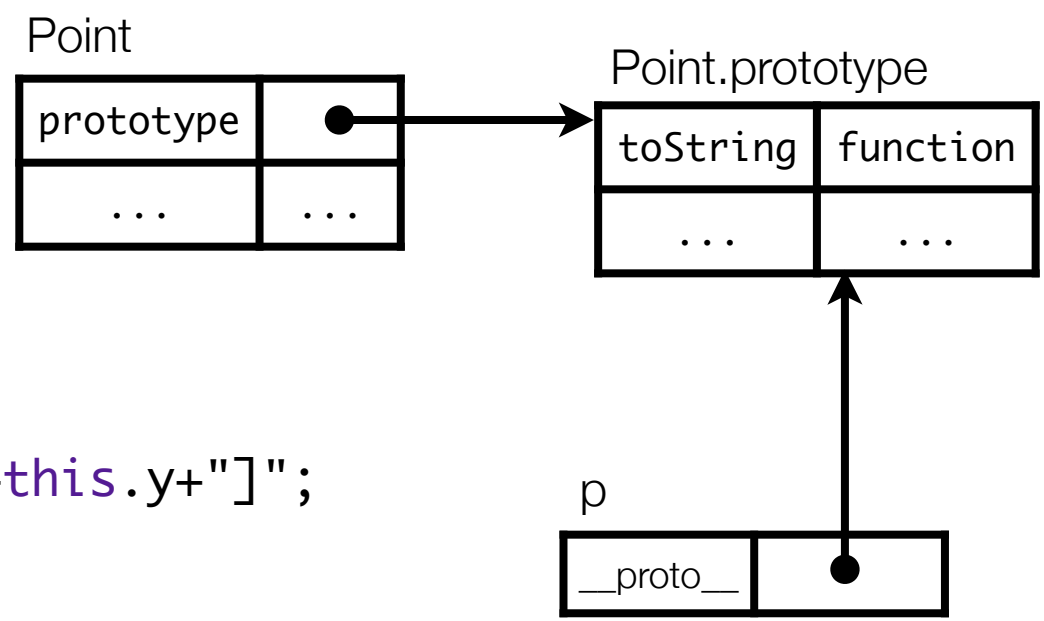
```
function add(x,y) { return x + y; }
add(1,2) // 3

add.doc = “returns the sum of two numbers”;
```

Objects

- No classes.
- Instead, functions may be used as object constructors.
- All objects have a “prototype” link
  - Lookup of a property on an object traverses the prototype links
  - Similar to inheritance between classes
  - In some implementations, the prototype is an explicit property of the object named `__proto__`

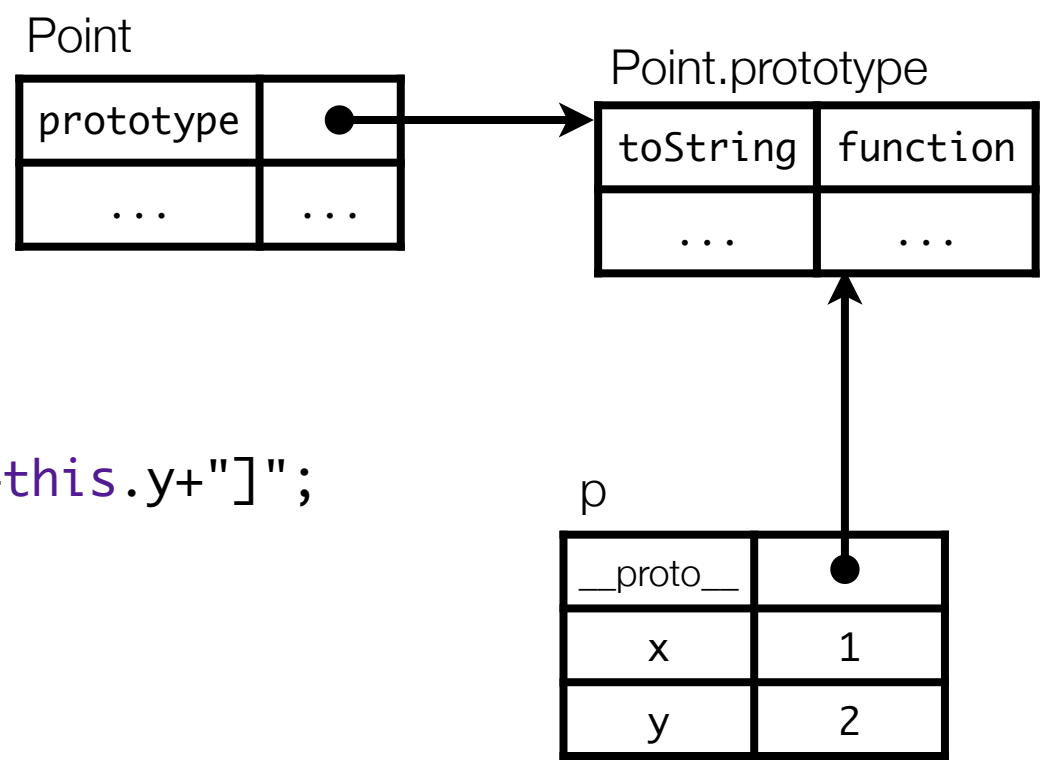
Objects

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

Point.prototype = {
  toString: function() {
    return "[Point "+this.x+","+this.y+"]";
  }
}

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



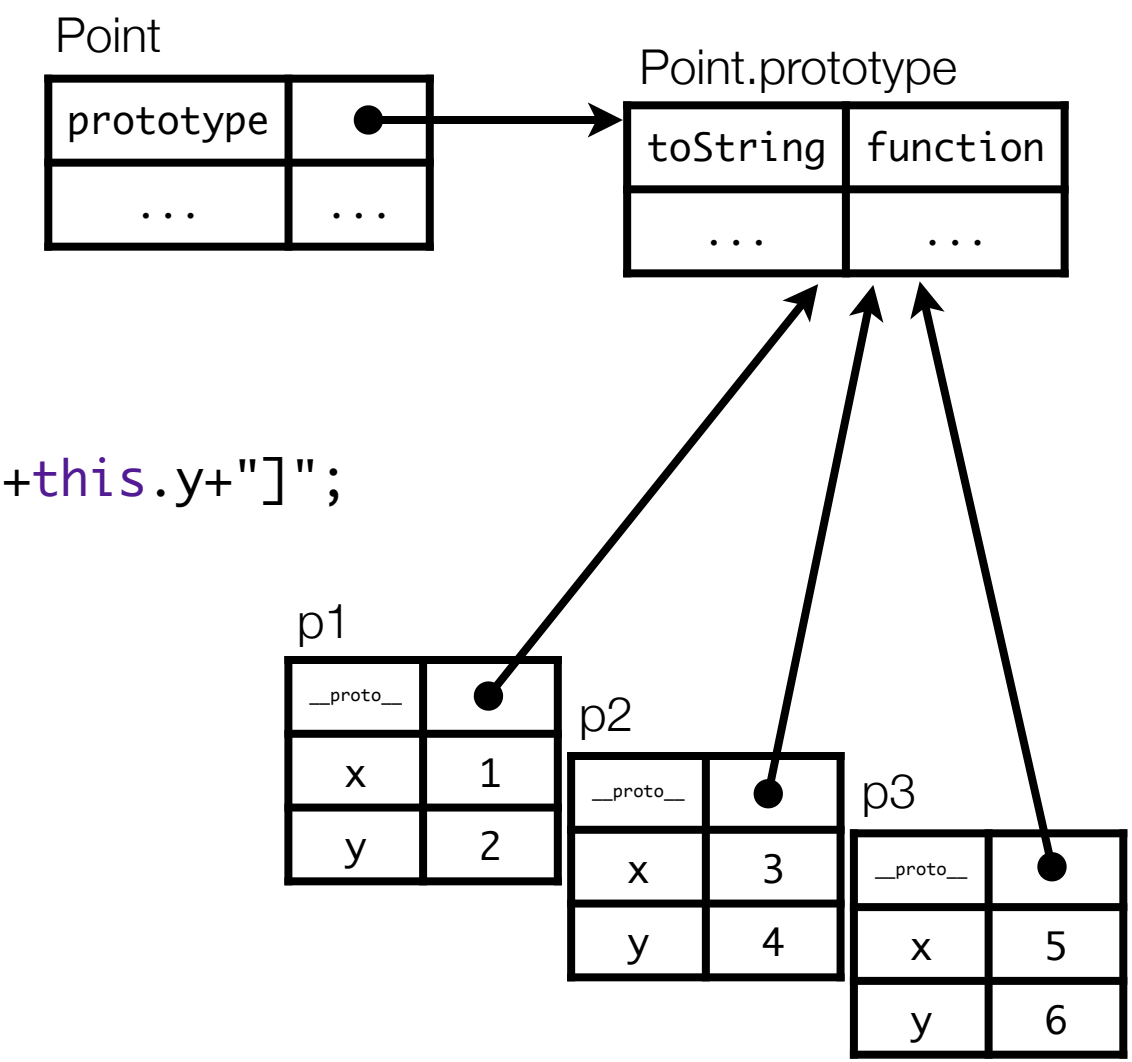
Objects

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

Point.prototype = {
  toString: function() {
    return "[Point "+this.x+","+this.y+"]";
  }
}

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



Objects

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

Point.prototype = {
  toString: function() {
    return "[Point "+this.x+","+this.y+"]";
  }
}

var p1 = new Point(1,2);
var p2 = new Point(3,4);
var p3 = new Point(5,6);
```



Functions versus Methods

- Methods of objects are just functions
- When a function is called “as a method”, `this` is bound to the receiver object

```
var obj = {
  offset: 10,
  index: function(x) { return this.offset + x; }
}

obj.index(0); // 10
```

Functions versus Methods

- Methods may be “extracted” from objects and used as stand-alone functions

```
var obj = {
  offset: 10,
  index: function(x) { return this.offset + x; }
}

var f = obj.index;

otherObj.index = f;

f(0) // error

f.apply(obj, [0]) // 10
```

Functions versus Methods

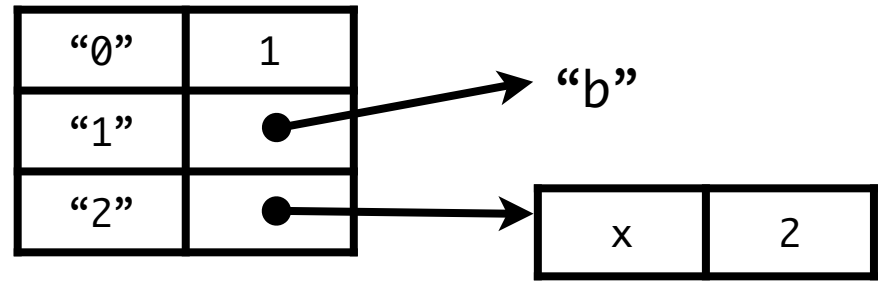
- Methods may be “extracted” from objects and used as stand-alone functions

```
var obj = {
  offset: 10,
  index: function(x) { return this.offset + x; }
}

var boundF = obj.index.bind(obj); // new in ES5

boundF(0) // 10
```

Arrays

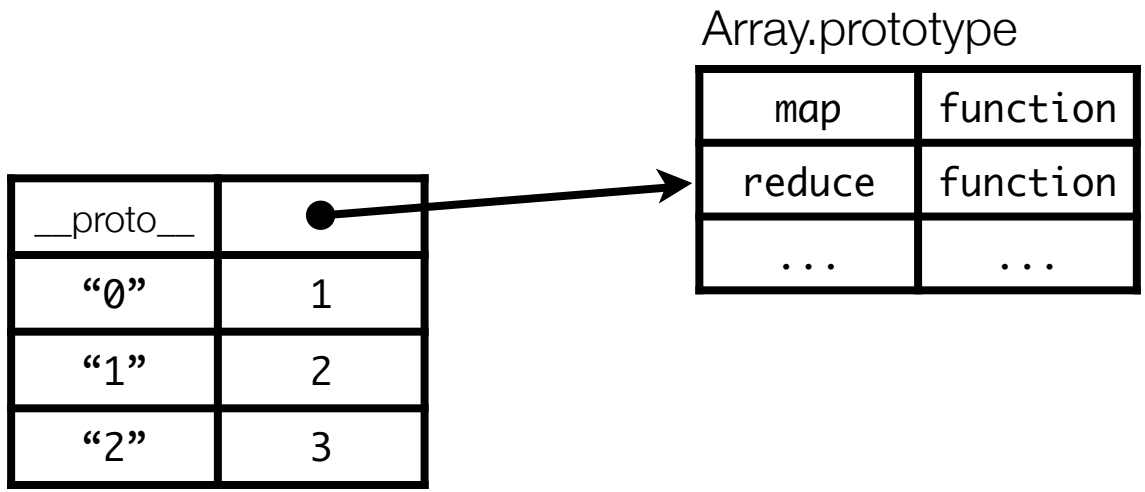
- Arrays are objects, too

```
var a = [1, "b", {x:2}];

a[0]        // 1
a["0"]      // 1
a.length    // 3
a[5] = 0;
a.length    // 6
a.foo = “test”
a.foo // “test”
```



Arrays

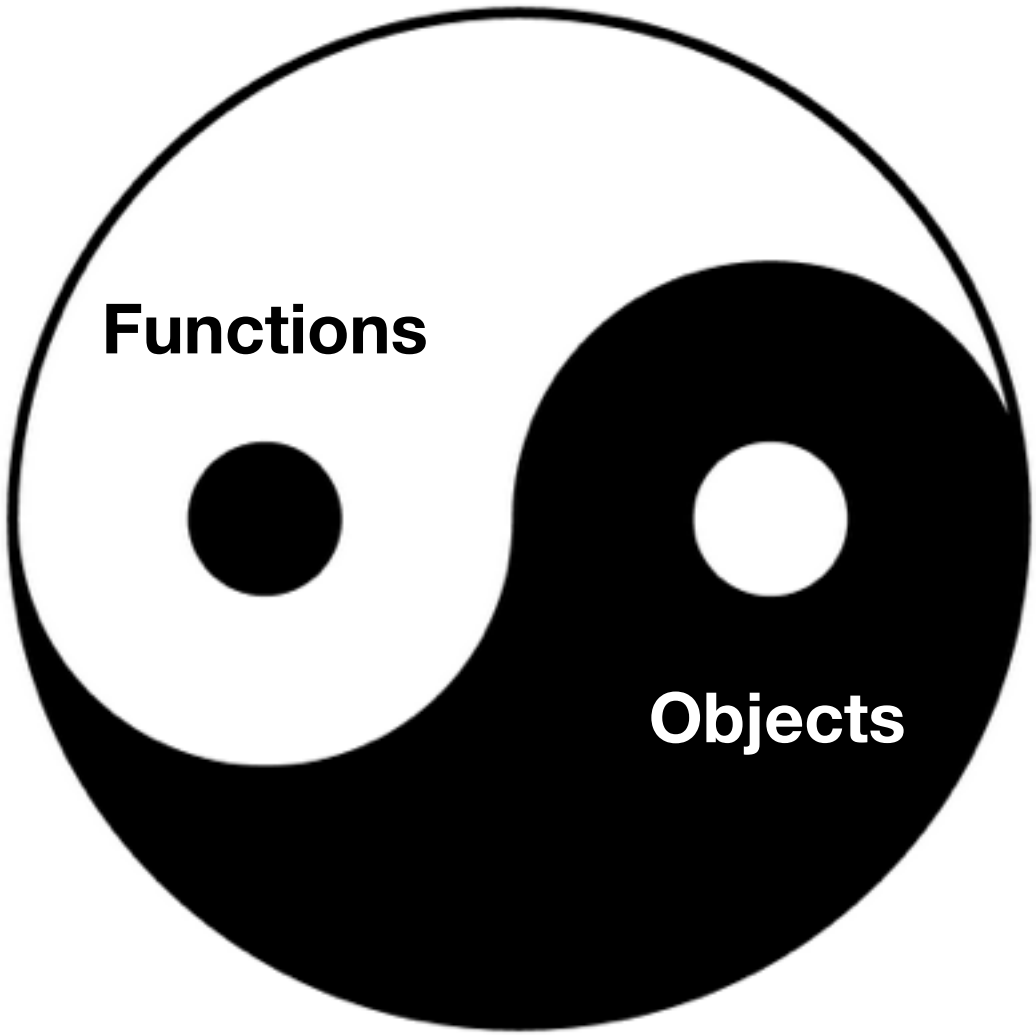
- Array instances inherit many useful functions from `Array.prototype`




```
[1,2,3].map(function (x) { return x + 1; }); // [2, 3, 4]

[1,2,3].reduce(function(sum, x) { return sum + x; }, 0); // 6

...
```

The Good Parts: summary

- Javascript: “a Lisp in C’s clothing” (D. Crockford)



Part II: Javascript, the **Bad** Parts

The Good Parts

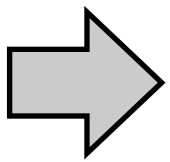
The Bad Parts

Bad Parts: global variables

- Scripts depend on global variables for linkage

Bad

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


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



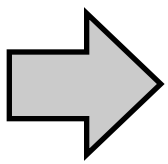
Better

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

Bad Parts: `with` statement

- `with`-statement turns object properties into variables

```
paint(widget.x,
      widget.y,
      widget.color);
```



```
with (widget) {
  paint(x,y,color);
}
```

Bad Parts: `with` statement

- `with`-statement breaks static scoping

```
var x = 42;
var obj = {};
with (obj) {
  print(x); // 42
  obj.x = 24;
  print(x); // 24
}
```

Bad Parts: var hoisting

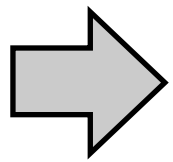
- JavaScript uses block *syntax*, but does not use block *scope*

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

Bad Parts: var hoisting

- 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>
```

Bad Parts: var hoisting

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

- Beware: nested loop indices

```
for (var i = 0; i < matrix.length; i++) {
  var row = matrix[i];
  for (var i = 0; i < row.length ; i++) {
      // …
  }
}
```

Bad Parts: var hoisting

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

- Beware: nested loop indices

```
var i, row;
for (i = 0; i < matrix.length; i++) {
  row = matrix[i];
  for (i = 0; i < row.length ; i++) {
      // assign to wrong i!
  }
}
```

Bad Parts: var hoisting

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

- Beware: functions within loops

```
var elts = document.getElementsByClassName("myClass");

for (var i = 0; i < elts.length; i++) {
  elts[i].addEventListener("click", function() {
    alert(i);
  });
}
```

Bad Parts: var hoisting

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

- Beware: functions within loops

```
var elts, i;
elts = document.getElementsByClassName("myClass");

for (i = 0; i < elts.length; i++) {
  elts[i].addEventListener("click", function() {
    alert(i);
  });
}
```

Bad Parts: var hoisting

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

- Beware: functions within loops

```
var elts = document.getElementsByClassName("myClass");

for (var i = 0; i < elts.length; i++) {
  (function() {
    var pos = i;
    elts[i].addEventListener("click", function() {
      alert(pos);
    });
  }())
}
```

Bad Parts: var hoisting

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

- Beware: functions within loops

```
var elts = document.getElementsByClassName("myClass");

for (var i = 0; i < elts.length; i++) {
  (function(pos) {
     elts[i].addEventListener("click", function() {
       alert(pos);
     });
   }(i))
}
```

Bad Parts: semicolon insertion

- Automatic semicolon insertion (ASI)

```
function meaningOfLife() {
  return;
    { answer: 42 };
}
```

Bad Parts: implicit type coercions

- + operator tries to do both addition and concatenation

- == operator coerces arguments before testing for equality

```
‘’ == ‘0’    // false
 0 == ‘’     // true
 0 == ‘0’    // true
```

- === operator does not coerce its arguments

```
‘’ === ‘0’    // false
 0 === ‘’     // false
 0 === ‘0’    // false
```

- Morale: avoid ==, always use ===

Controversial Parts: eval

```
var obj = { a: 42 };
var name = “a”;
eval(“v = obj.” + name + “;”);
v // 42
```

“eval is Evil: The eval function is the single most misused feature of JavaScript.”

(Crockford in JS: The Good Parts)

Controversial Parts: eval

```
var obj = { a: 42 };
var name = "a";
var v = obj[name];
v // 42
```

"eval is Evil: The eval function is the single most misused feature of JavaScript."

(Crockford in JS: The Good Parts)

Bad Parts: typeof

- typeof operator returns the “type” of a value as a string

```
typeof 1              // “number”
typeof true           // “boolean”
typeof “hello”        // “string”
typeof undefined      // “undefined”
typeof {}             // “object”
typeof function(){} // “function”
typeof []             // “object” /!\
typeof null           // “object” /!\
```

- To test if x is an array, use `Array.isArray(x)`

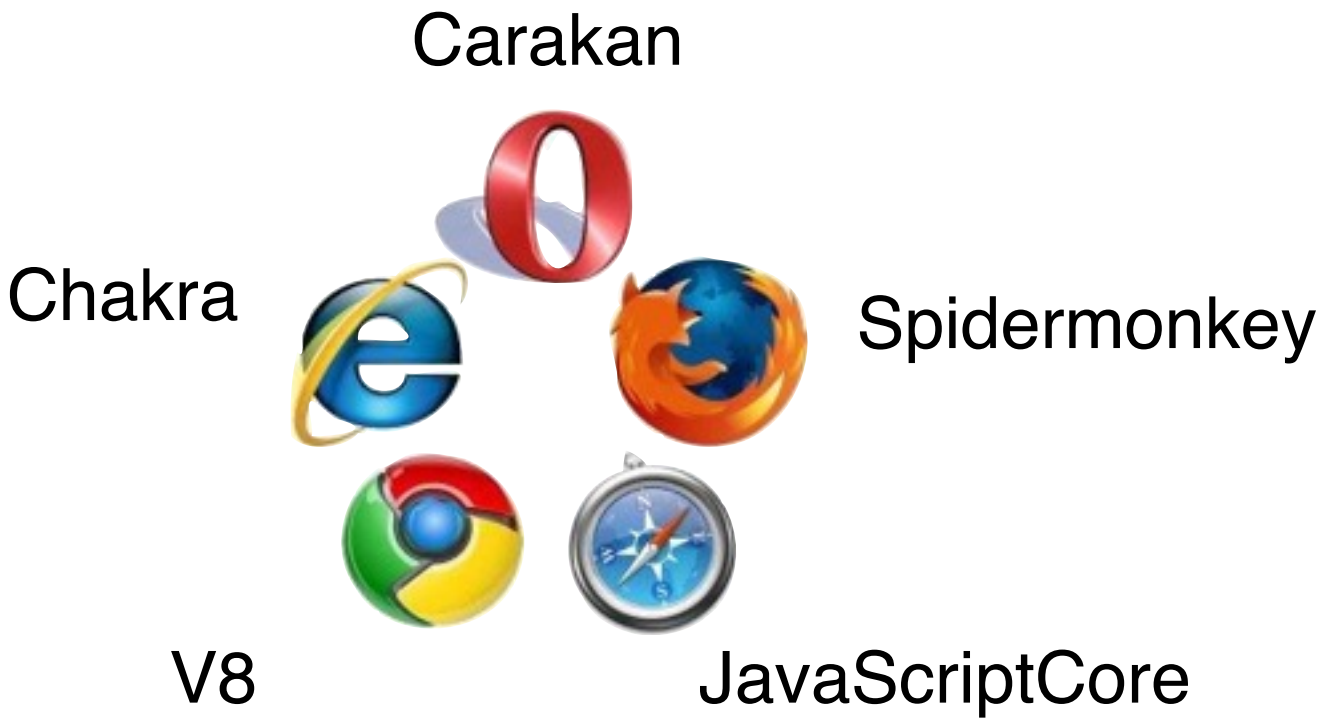
- To test if x is null, use `x === null`

The Bad Parts: summary

- Javascript: looks can be deceiving
- Globals: top-level variables *appear* local to script, but are global
- Hoisting: variables *appear* block-scoped, but are function-scoped
- Semicolon insertion is *invisible*
- Certain operators *implicitly* coerce arguments

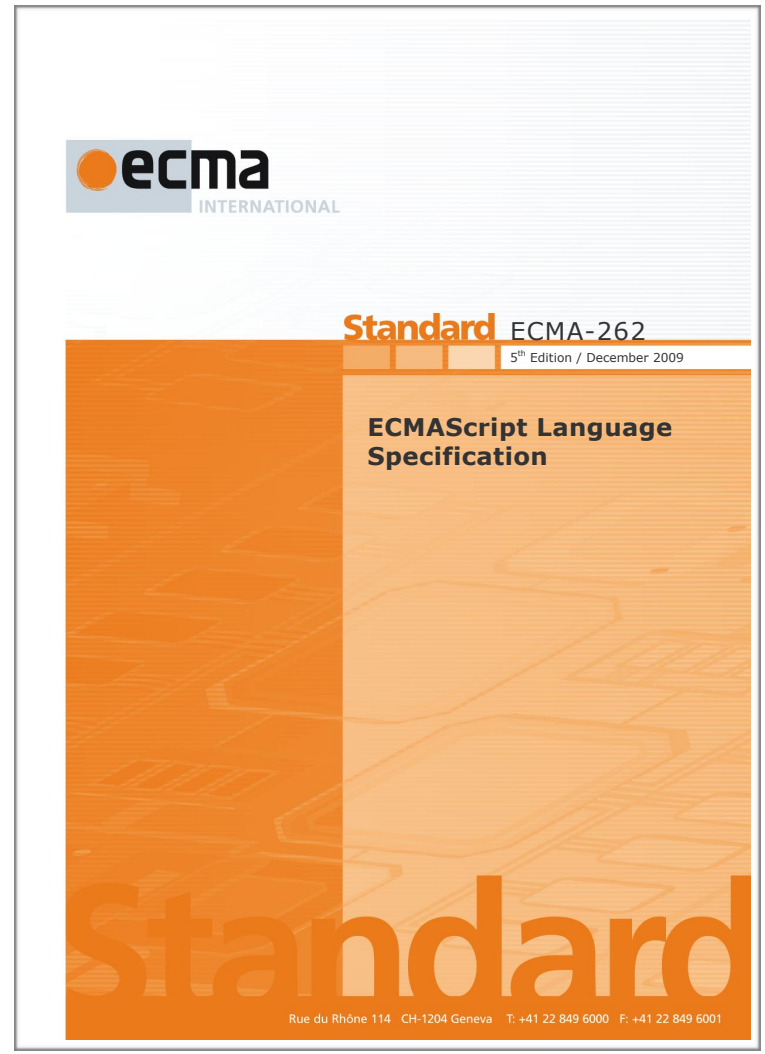
Part III: ECMAScript 5 and the **Strict** Parts

ECMAScript: “Standard” JavaScript



ECMAScript specification

- 1st ed. 1997
- 2nd ed. 1998
- 3rd ed. 1999
- ~~4th ed.~~
- 5th ed. 2009
- *6th ed. end of 2014 (tentative) (draft is available)*



ECMAScript 5 Themes

- New APIs, including JSON
- Support for more robust programming
  - Tamper-proof objects
  - Strict mode

ECMAScript 5 Themes

- **New APIs, including JSON**
- Support for more robust programming
  - Tamper-proof objects
  - Strict mode

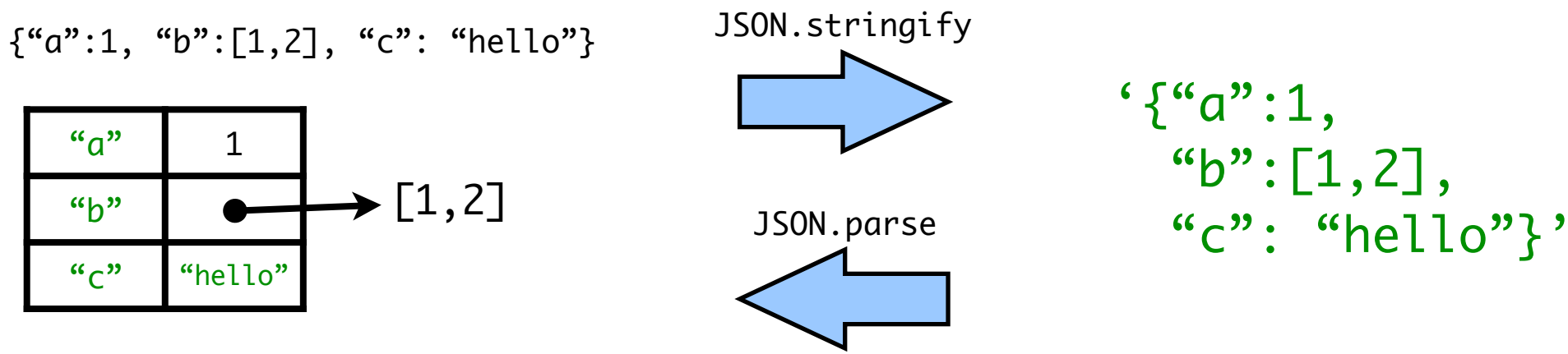
JSON

- **J**ava**S**cript **O**bject **N**otation
- A subset of Javascript to describe *data* (numbers, strings, arrays and objects without methods)
- Formal syntax literally fits *in a margin*. See http://json.org/

```
{ “name” : “Bob”,
  “age” : 42,
  “address” : {
    “street” : “Main st.”
  }
}
```

ECMAScript 5 and JSON

- Before ES5, could either parse quickly or safely
- Unsafe: `eval(jsonString)`
- In ES5: use `JSON.parse`, `JSON.stringify`



ECMAScript 5 Themes

- New APIs, including JSON
- **Support for more robust programming**
  - Tamper-proof objects
  - Strict mode

Tamper-proof Objects: motivation

- Objects are *mutable* bags of properties
- Cannot protect an object from modifications by its clients
- External clients may *monkey-patch* shared objects
  - Powerful: allows to fix bugs or extend objects with new features
  - Brittle: easily leads to conflicting updates

Tamper-proof Objects

```
var point =
  { x: 0,
    y: 0  };


Object.preventExtensions(point);
point.z = 0; // error: can't add new properties


Object.seal(point);
delete point.x; // error: can't delete properties


Object.freeze(point);
point.x = 7; // error: can't assign properties
```

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)

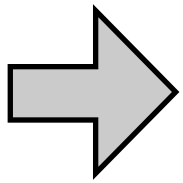
```
<script>
"use strict";
...
</script>
```

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

Strict-mode opt-in: gotcha's

- Beware: minification and deployment tools may concatenate scripts

```
<script>
"use strict";
// in strict mode
</script>

<script>
// not in strict mode
function f(){...}
</script>
```

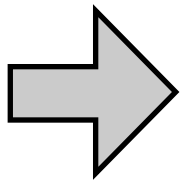


```
<script>
"use strict";
// in strict mode



// f is now
// accidentally strict!
function f(){...}
</script>
```

Strict-mode opt-in: gotcha's

- Suggested refactoring is to wrap script blocks in function bodies

```
<script>
(function(){
  "use strict";
  // in strict mode
}())
</script>
```



```
<script>
(function(){
  "use strict";
  // in strict mode
}())
```

```
<script>
// not in strict mode
function f(){...}
</script>
```

```
// not in strict mode
function f(){...}
</script>
```

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!)
```

Direct versus Indirect Eval

- ES5 runtime changes to eval (both in strict and non-strict mode)
- `eval` “operator” versus `eval` “function”

Direct Eval

```
function f() {
  var x = 0;
  eval(“x = 5”);
  return x;
}
f() // returns 5
```

Indirect Eval

```
function f(g) {
  var x = 0;
  g(“x = 5”);
  return x;
}
f(eval) // returns 0
```

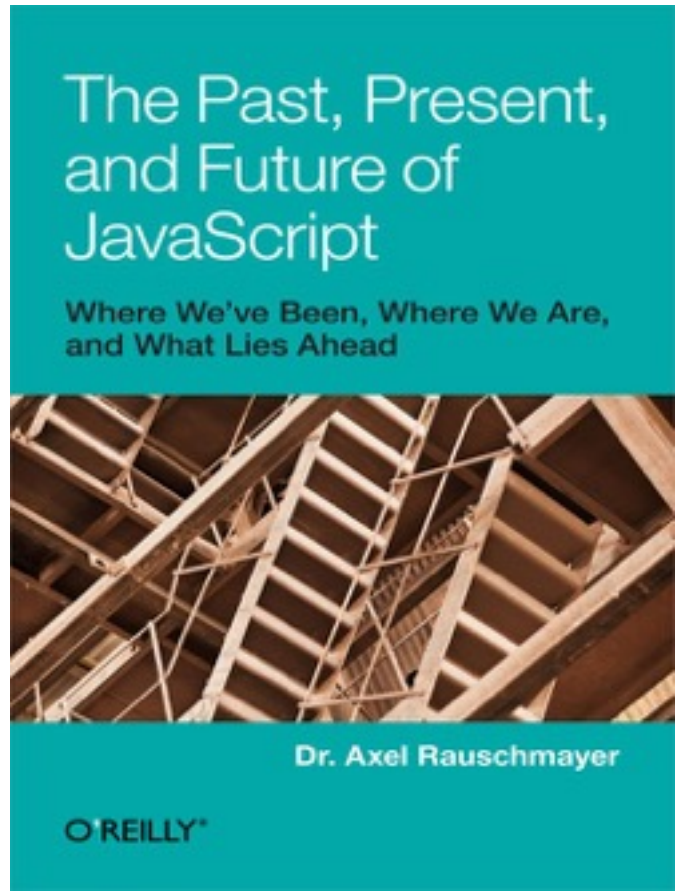
ECMAScript 5 Themes: summary

- New APIs, including JSON
- Support for more robust programming
  - Tamper-proof objects
  - Strict mode

Part IV: ECMAScript 6, the **New** Parts

ECMAScript 6

- Many new additions (too many to list here *)
- Arrow functions, rest and optional args
- Classes
- Modules
- String templates
- Proxies



* see http://kangax.github.io/es5-compat-table/es6/ for an overview of ES6 features

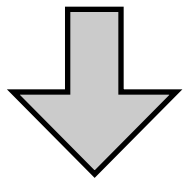
ECMAScript 6: improving functions

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

ECMAScript 6: arrow functions

ES5

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



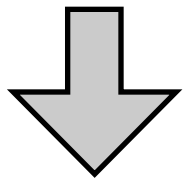
ES6

```
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);
  …
}
```



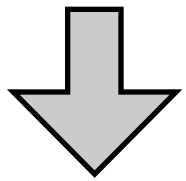
ES6

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

ECMAScript 6: optional arguments

ES5

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



ES6

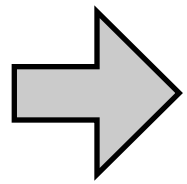
```
function greet(name = “world”) {
  return “Hello, ” + name;
}
```

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];
```



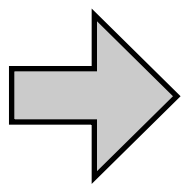
ES6

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

ECMAScript 6: 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...]";
  }
}
```



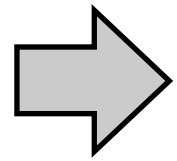
```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  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>
```



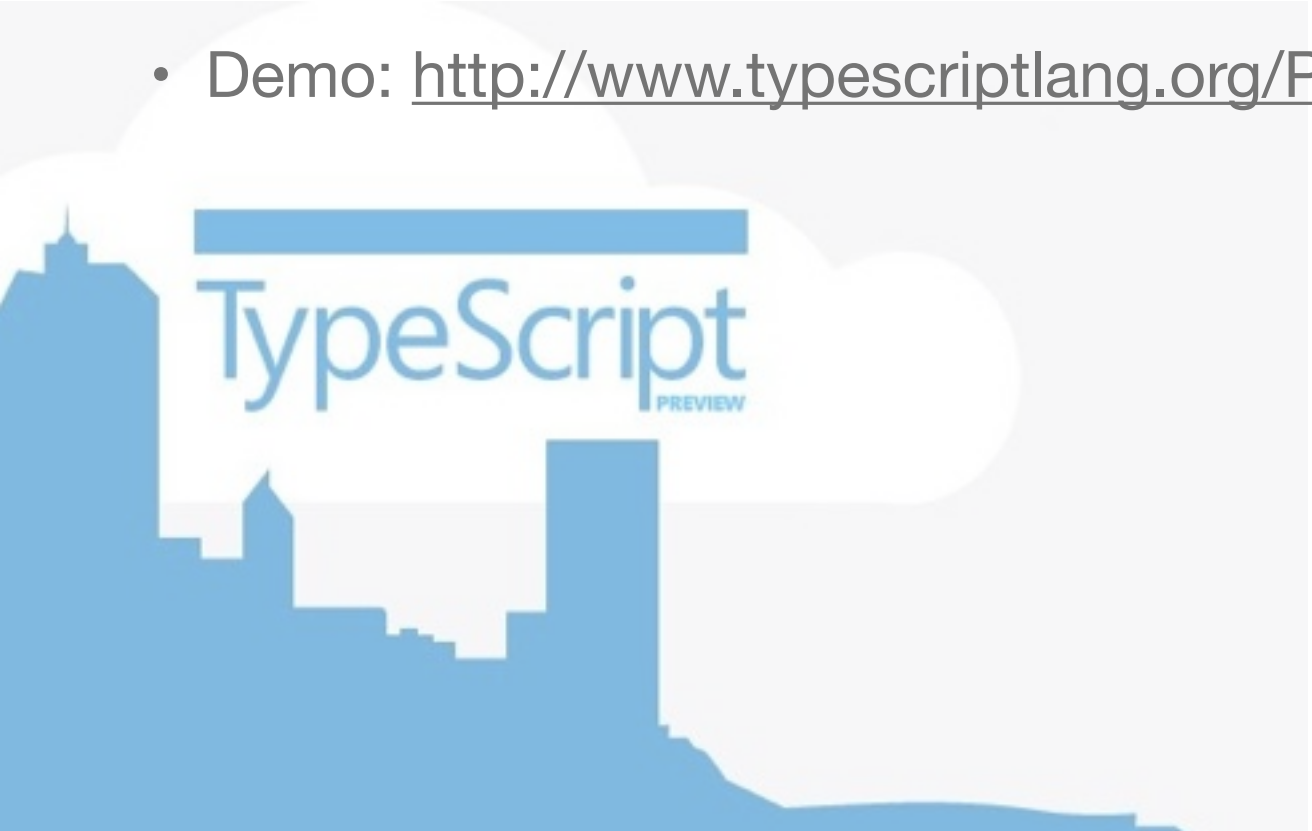
```
<script type="module"
        name="myLib">
var x = 0; // local!
export function inc() {
  return ++x;
}
</script>
```

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

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

TypeScript

- Can use classes, modules and arrow functions today in TypeScript
- Bonus: type inference
- Demo: http://www.typescriptlang.org/Playground/



ECMAScript 6 string templates

- 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 string templates

- String templates 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 string templates

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

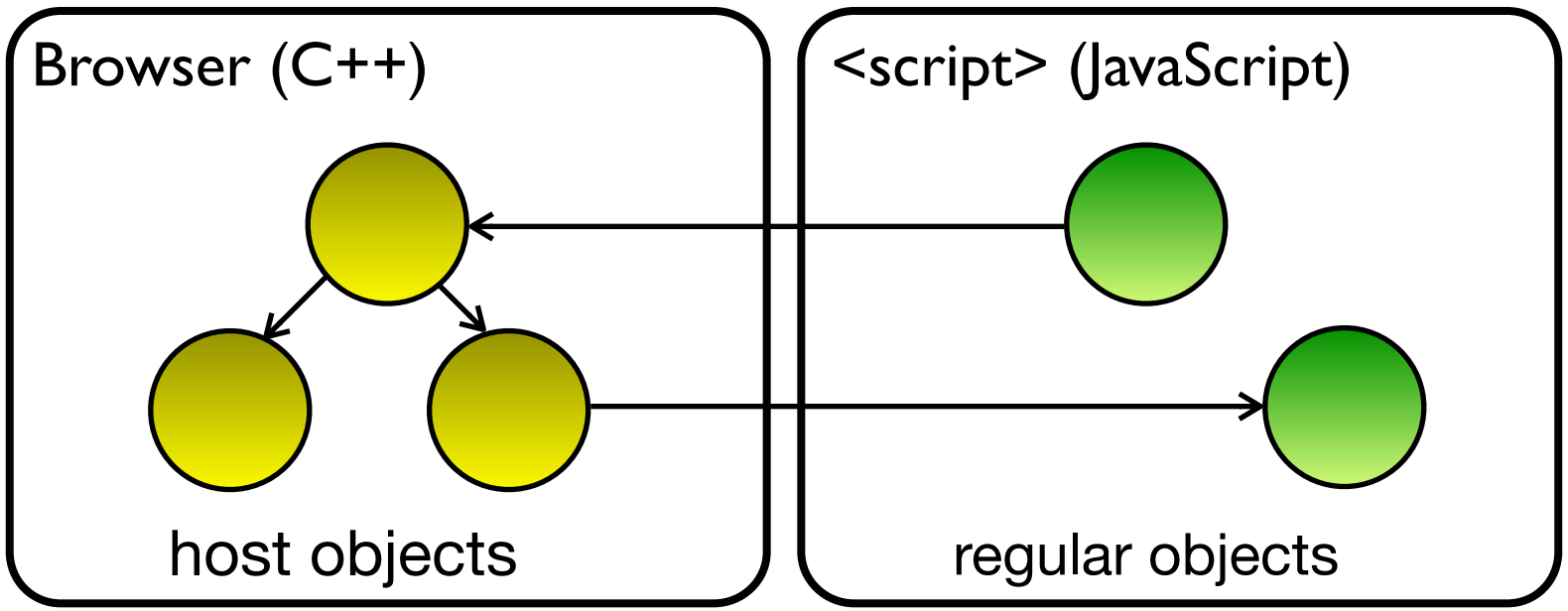
```
function createDiv(input) {
  return html(["<div>","</div>"], input);
};

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

ECMAScript 6 proxies

- Dynamic proxy objects: objects whose behavior can be controlled in JavaScript itself
- Goals:
  - Create generic object wrappers
  - Emulate host objects

Host objects

- Objects provided by the host platform
- E.g. the **DOM**: a tree representation of the HTML document
- Appear to be Javascript objects, but not implemented in Javascript
- Can have odd behavior that regular JavaScript objects cannot emulate



Proxy example: log all property accesses

```
function makePoint(x, y) {
  return {
    x: x,
    y: y
  };
}
```

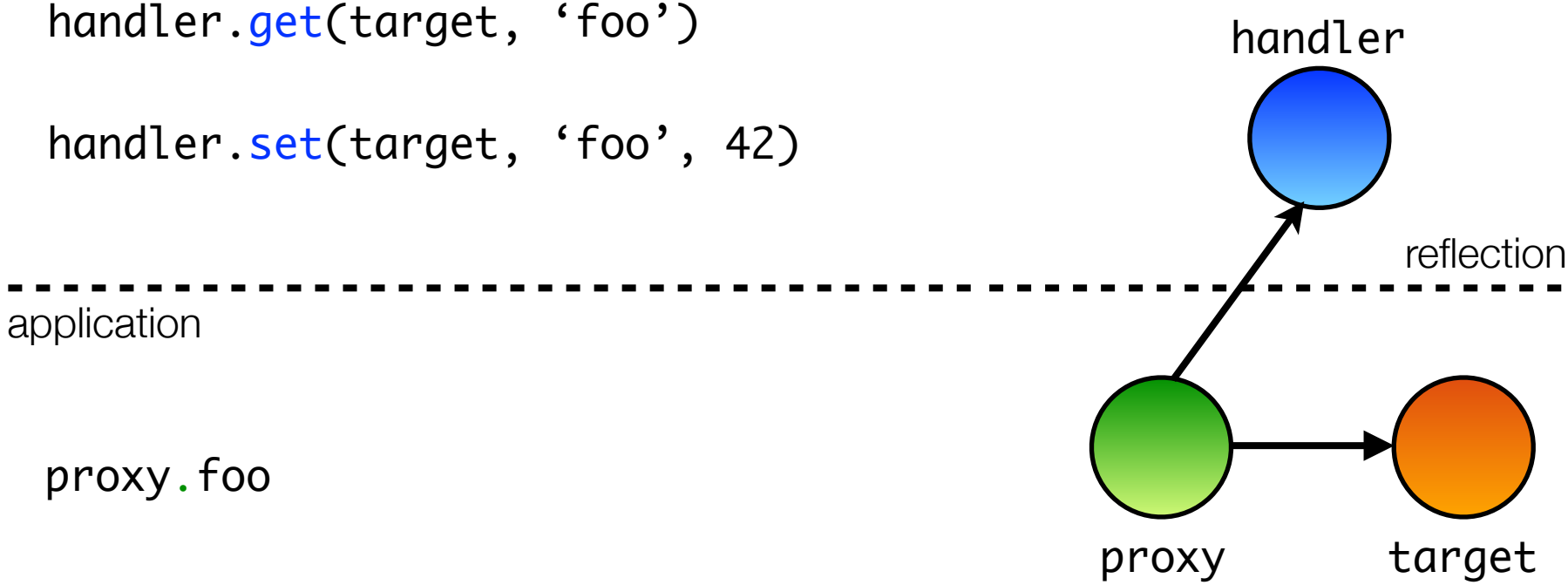
```
var p = makePoint(2,2);
var lp = makeLogger(p);
lp.x
// log: get x
// returns 2
lp.y = 3
// log: set y 3
```

ECMAScript 6 proxies

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

```
handler.get(target, 'foo')
```

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




```
proxy.foo
```

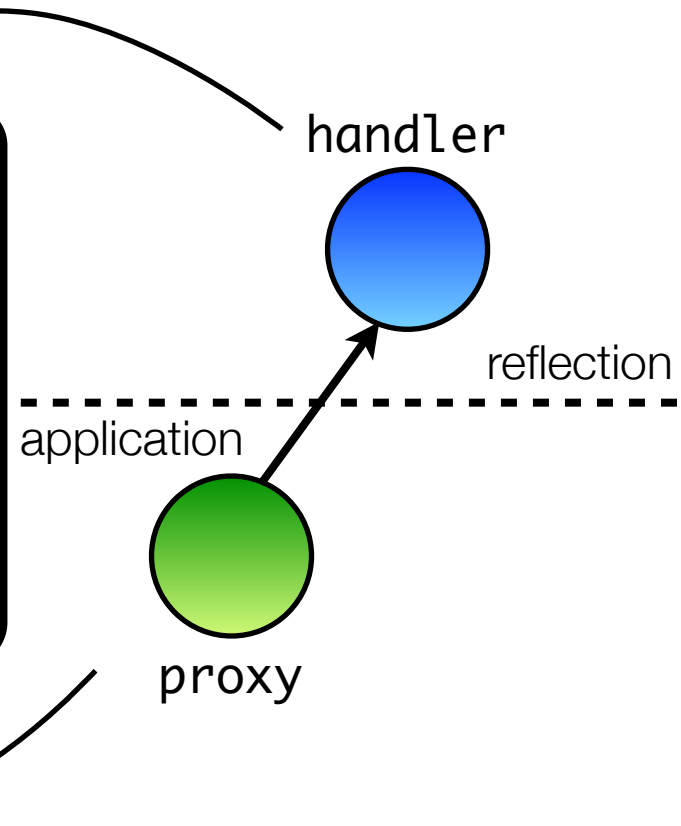
```
proxy.foo = 42
```

Example: logging all property accesses

```
function makeLogger(target) {
  var proxy = new Proxy(target, {
    get: function(target, name) {
      console.log('get', name);
      return target[name];
    },
    set: function(target, name, val) {
      console.log('set', name, val);
      return target[name] = val;
    },
  });
  return proxy;
}
```

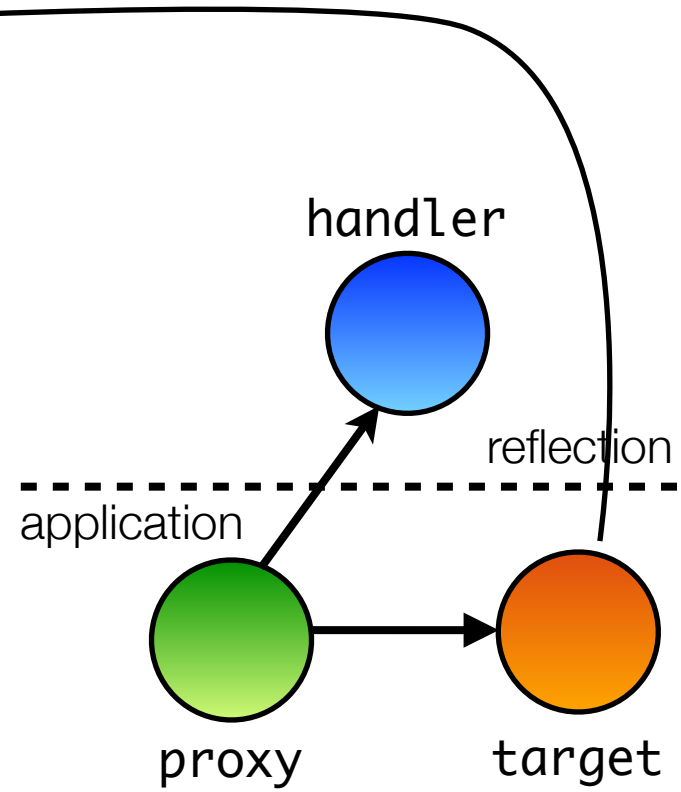
Example: logging all property accesses

```
function makeLogger(target) {
  var proxy = new Proxy(target, {
    get: function(target, name) {
      console.log('get', name);
      return target[name];
    },
    set: function(target, name, val) {
      console.log('set', name, val);
      return target[name] = val;
    },
  });
  return proxy;
}
```



Example: logging all property accesses

```
function makeLogger(target) {
  var proxy = new Proxy(target, {
    get: function(target, name) {
      console.log('get', name);
      return target[name];
    },
    set: function(target, name, val) {
      console.log('set', name, val);
      return target[name] = val;
    },
  });
  return proxy;
}
```



Wrap-up

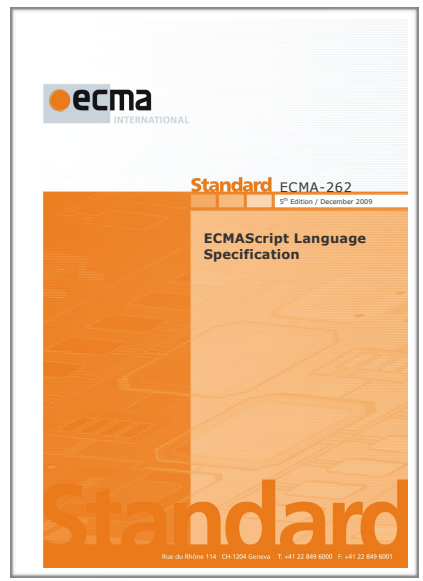
Wrap-up

JavaScript:

the **Good**,

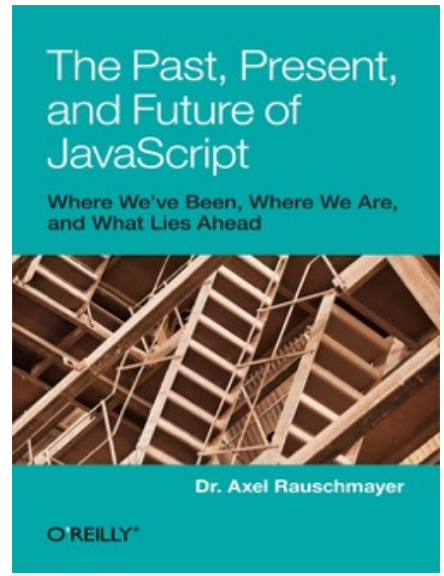
the **Bad**,

the **Strict**,



and

the **New** Parts.



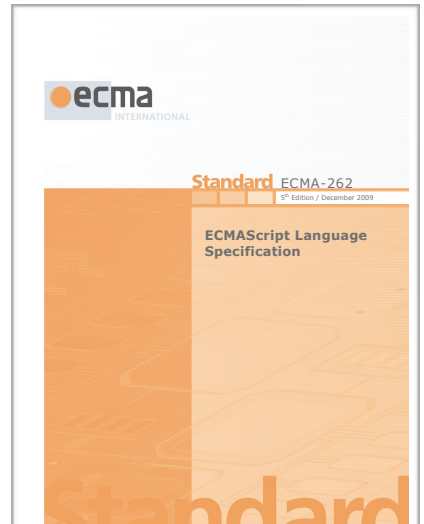
Take-home messages

- JavaScript: a Lisp in C's clothing
- A simple core language that is both good at OOP and FP

the Good,

the Bad,

the Strict,



and
the New P

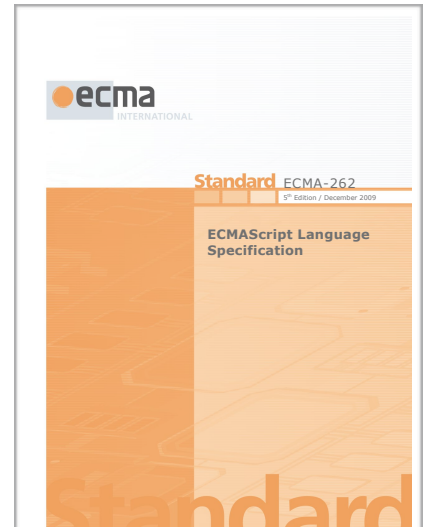
Take-home messages

- JavaScript has “warts” that can’t be easily removed
- Mastering JavaScript involves learning to avoid the bad parts

the Good,

the Bad,

the Strict,



and
the New P

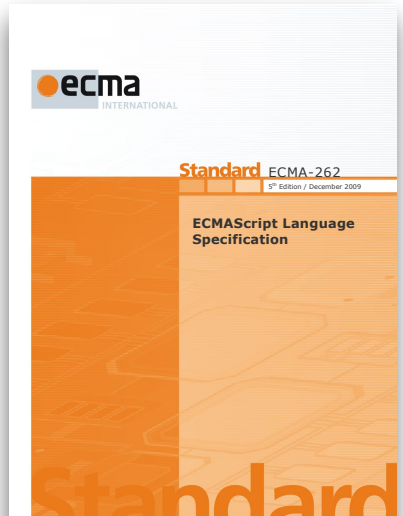
Take-home messages

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

the Good,

the Bad,

the Strict,



and
the New P

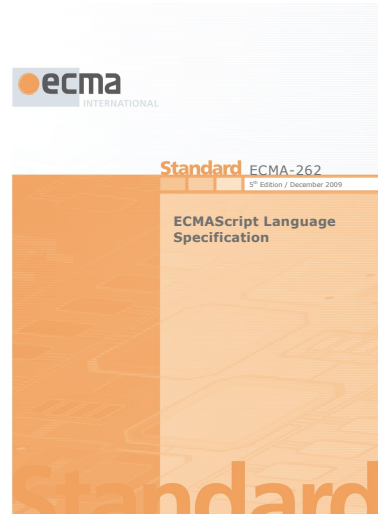
Take-home messages

- ECMAScript 6 will introduce many new features
- Will help to further move away from the bad parts

the Good,

the Bad,

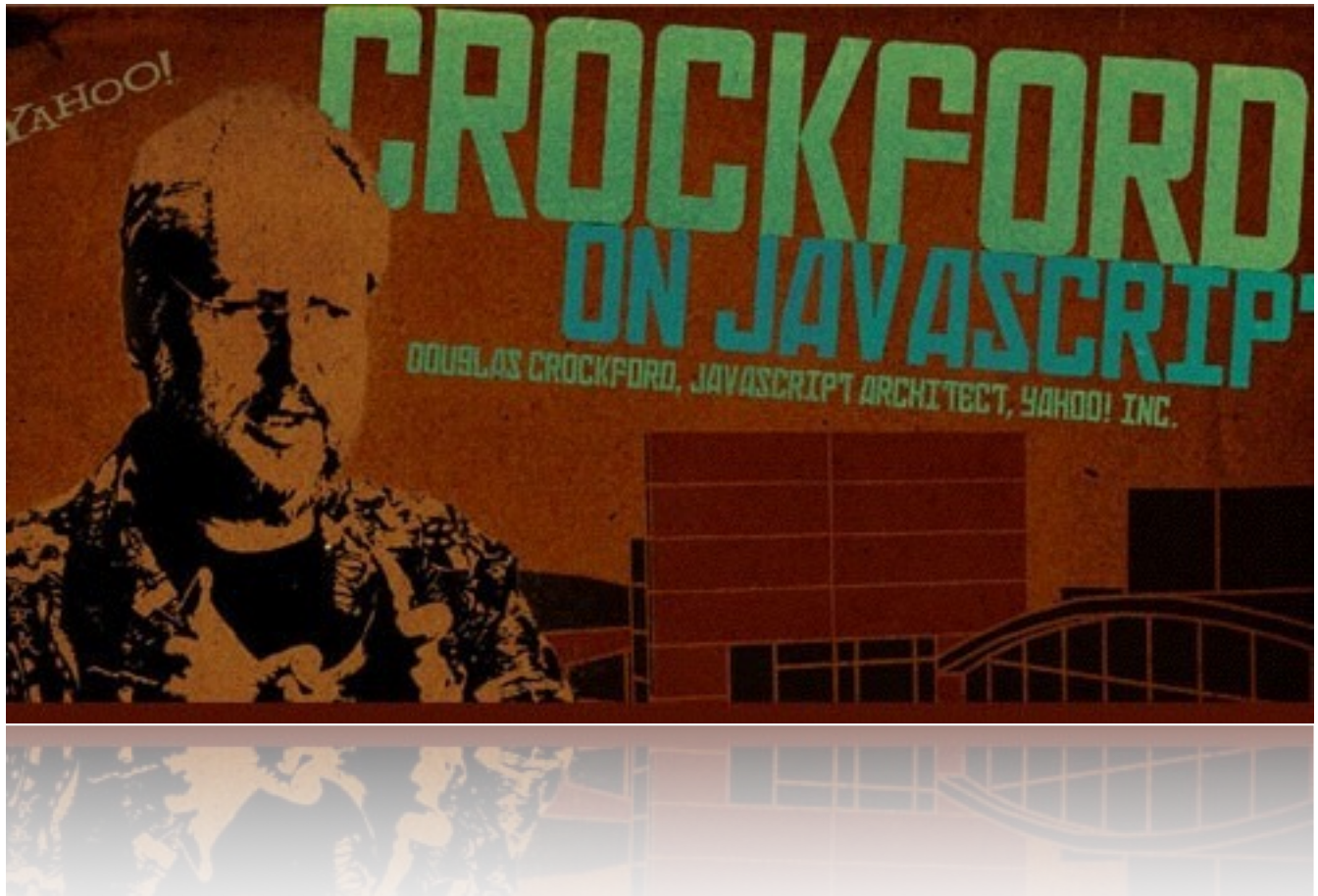
the Strict,



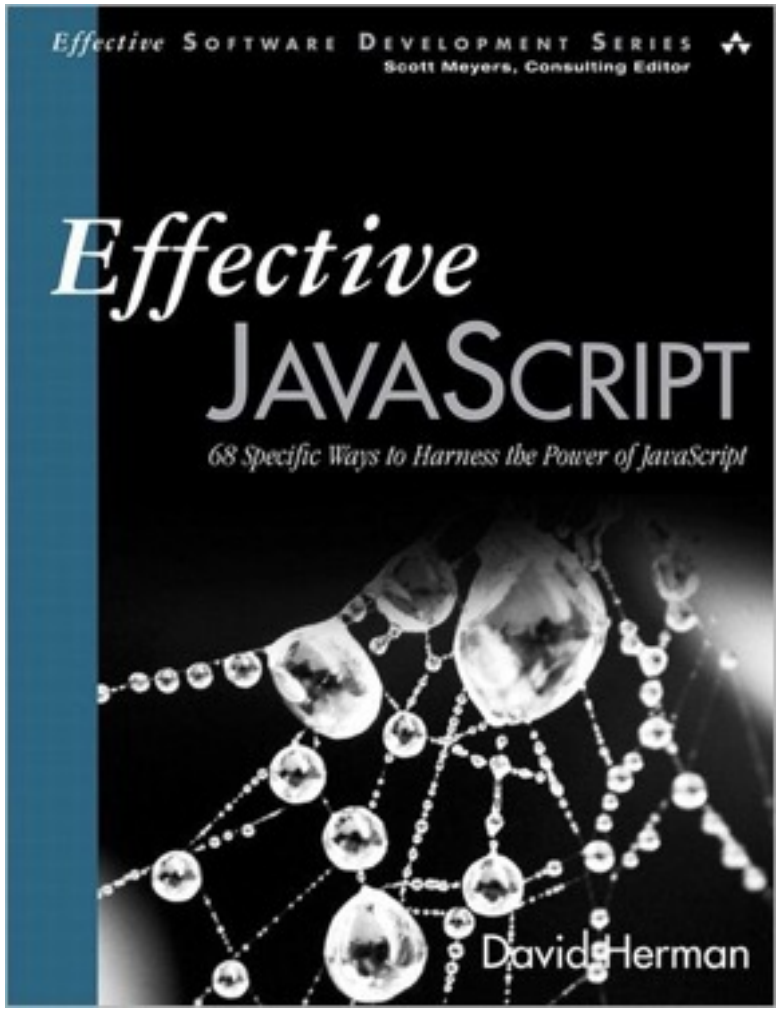
and
the New Parts

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

Thanks for listening!

JavaScript: the Good, the Bad, the Strict and the New Parts

Tom Van Cutsem