Changes to JavaScript, Part 1: EcmaScript 5

Mark S. Miller, Waldemar Horwat, Mike Samuel
Your EcmaScript committee representatives

Brief History

EcmaScript stuck at ES3 since 1999
- Several attempts at feature "rich" ES4.

Successful ES3.1 simplification, renamed ES5
- IE, FF implementations almost beta.
- Progress at Opera,v8,⋯. Not yet Safari.

meld(ADsafe,Caja,⋯) → *Secure EcmaScript*
- Four synergies between SES and ES5

ES-Harmony agreement on future directions
- Lexical scoping, classes as sugar

EcmaScript 5 Overview

Theme: Emulate Platform (DOM) Objects

- Accessor (getter / setter) properties
- Property attribute control

Theme: More Robust Programming

- Tamper-proof objects
- Wart removal, clarifications, and exchanges
- Strict mode

Theme: New APIs

- Higher order array methods
- JSON
- Function.prototype.bind

EcmaScript 5 Overview

- Theme: Emulate Platform (DOM) Objects
  - Accessor (getter / setter) properties
  - Property attribute control
- Theme: More Robust Programming
  - Tamper-proof objects
  - **Wart removal, clarifications, and exchanges**
  - Strict mode
- Theme: New APIs
  - Higher order array methods
  - JSON
  - Function.prototype.bind

Aren't standards doomed to grow?

New standard must be *practically* upward compatible

- Not from ES3, but from cross-browser web corpus.
- Annoying differences → opportunity for committee.
- 3 of 4 browser rule + experience & sense.

Aren't standards doomed to grow?

- New standard must be *practically* upward compatible
  - Not from ES3, but from cross-browser web corpus.
  - Annoying differences → opportunity for committee.
  - 3 of 4 browser rule + experience & sense.
- Opt-in for strict code: `"use strict";`
  - Mostly subtractive!
  - Better for new code, for novices & experts.
  - Not purely compatible, but good code ports easily.

Problem:
Can’t emulate platform objects

```
domObject.innerHTML = …;

Object.prototype.myToString = …;
for (var k in {}) {
  //Sees ‘myToString’ but not ‘toString’.
}
```

Primitives have powers unavailable in JS.

Can’t transparently emulate or wrap.

Accessor (getter/setter) properties

`domNode.innerHTML`

ES3 can't create properties which compute on assignment or reading.

Mozilla, others, provide getter/setter properties, though with divergent semantics.

Popular, useful. Codify proven practice.

Accessor (getter/setter) properties…

…by enhanced object literal:

```
var domoid = {
    foo: ‘bar’,
    get innerHTML() { return …; },
    set innerHTML(newHTML) { … }
};
```

…by explicit control (later)…

Need property attribute control

ES3 semantics define per-property attributes:

- ReadOnly → `writable:` does assignment work?
- DontEnum → `enumerable:` does for-in see it?
- DontDelete → `configurable:` can its shape change?

ES3 code can only create properties that are writable, enumerable, and deletable.

Now controllable by ES5 code.

Property attribute control API

```
Object.create(p,{(n:attrs)*})
  .defineProperty(o,n,attrs)
  .defineProperties(o,{(n:attrs)*})
  .getOwnPropertyNames(o) → names
  .getOwnPropertyDescriptor(o,n) → attrs
  .freeze(o) → o

attrs ::= { value:v,
            writable:b,
            enumerable:b,
            configurable:b } | …
```

Example: Unobtrusive Convenience Methods

```
Object.defineProperty(Object.prototype,
                      'myToString', {
  value: …,
  writable: true,
  enumerable: false,
  configurable: true
}


for (var k in {}) {
  //Doesn't see 'myToString' or 'toString'
}
```

Accessor (getter/setter) properties…

…by enhanced attribute control API

```
Object.defineProperty(domoid,
                      'innerHTML',{
  get: function() { return …; },
  set: function(newHTML) { … }
});
```

attrs ::= … | { get:*f()→v*,
set:*f(v)*,
enumerable:*b*,
configurable:*b* }

Practical emulation of host objects

Property attribute control
+ Accessor (getter/setter) properties
= Able to emulate DOM API in ES code

Puts library developers more on par with platform (browser) providers

ES3 Makes Robustness Too Hard

- ES3 is full of traps and minefields.
- Defensive programming impossible
  - Objects are merely mutable maps of properties.
  - Silent errors. *Did this assignment really happen?*
  - Even lexical closures aren't quite encapsulated.
- Toxic leaks of *the global object* (`window`)

More Robust Programming in ES5

Tamper proof objects

- Objects can be sealed or frozen.
- Primordial objects should be frozen after initialization.

Wart removal, clarification, and exchanges

Defensive "strict" code

- Noisy errors - assumption violations throw.
- Static scoping. Functions really encapsulated.
- No implicit wrapping. No global object leakage.

What's the Point?
Can't preserve invariants in ES3

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

var pt = new Point(3,5);
pt.x = 'foo'; //Clobbers pt's consistency
```

Modular oo programming:

- Clients make requests of providers.
- Provider's responsibility to maintain own invariants.

Tamper proof objects

```
Object.create(p,{(n:attrs)*})
  .defineProperty(o,n,attrs)
  .defineProperties(o,{(n:attrs)*})
  .getOwnPropertyNames(o) → names
  .getOwnPropertyDescriptor(o,n) → attrs
  .freeze(o) → o

attrs ::= { value:v,
            writable:b,
            enumerable:b,
            configurable:b } | …
```

Tamper proof points, first try

```
function Point(x, y) {
  return Object.freeze({
    x: +x,
    y: +y
  });
}
```

But

Doesn't inherit methods from `Point.prototype`.

```
(new Point(3,4) instanceof Point) === false
```

When adequate, stop here.

Tamper proof points, second try

```
function Point(x, y) {
  return Object.freeze(
           Object.create(Point.prototype,
  {
    x: { value: +x, enumerable: true },
    y: { value: +y, enumerable: true }
  }));
}
```

Correct. But verbose boilerplate.

Frozen object builder helper

```
Object.defineProperty(Function.prototype,
                      'build', {
  value: function(attrMap) {
    return Object.freeze(Object.create(
      this.prototype, attrMap));
  },
  writable: false,
  enumerable: false,
  configurable: false
});

for (var k in n) … // not hurt
```

Definition fails if `Function.prototype` frozen.

Tamper proof points

```
function Point(x, y) {
  return Point.build({
    x: {value: +x, enumerable: true},
    y: {value: +y, enumerable: true}
  });
}
```

```
var pt = new Point(3,5);
pt.x = 'foo'; // fails, as it should
new Point(3,5) instanceof Point === true
    Point(3,5) instanceof Point === true
```

Scoping accident 1: Nested named function expressions

```
function foo() {
    var x = 8;
    bar(function baz() { return x; });
}
Object.prototype.x = 'foo';
```

Which **x** does "`return x`" return?

Scoping accident 1: Nested named function expressions

```
function foo() {
    var x = 8;
    bar(function baz() { return x; });
}
Object.prototype.x = 'foo';
```

Which **x** does "`return x`" return?

ES3: `'foo'`.

Editorial error deployed by some browsers.

ES5: Statically scoped where possible, so `8`.

Finding Opportunity in Confusion

When ES3 spec didn't say what was meant,

- Some browsers obeyed letter. Some obeyed saner spirit.

So cross-browser web is compatible with saner.

ES5 can then be saner.

Scoping accident 2: Throwing a function

```
function foo() { this.x = 11; }
var x = 'bar';
try { throw foo; } catch (e) {
  e();
  … x …
}
```

ES3: Another editorial error deployed.

ES5: Normal static scoping again.

Scoping accident 3: “as if by” initial or current?

```
Object = Date;
x = {}; //Assign to x a new what?
```

Scoping accident 3: "as if by" initial or current?

```
Object = Date;
x = {}; //Assign to x a new what?
```

ES3 spec language:

"as if by the expression '`new Object()`' "

ES5 clarifies: *initial* values are meant.

Wart exchange: keyword collision

ES3: `domNode.className`

- But html attribute is "`class`".
- Renamed to avoid collision with keyword

ES5: *`<expression>`*`.`*`<any-ident>`*

`{` *`<any-ident>`*`:` *`<expression>`* `…` `}`

- Covers simple JSON property use.
- No syntactic ambiguity with keywords.
- Doesn't fix DOM, but stops digging deeper hole.
- OTOH, complicates grammar.

Wart removal: Backchannel

```
function foo() { return (/x/); }
```

ES3: `foo() === foo() //Shared mutable state`

ES5: `(/x/) ≡ new RegExp('x')`

Wart exchange: eval in what scope?

```
function foo(a, b, c) {
    return a(b);
}

foo(eval, 'c', 37);
```

ES3: Returns 37? unclear

ES5: An *indirect eval*. Uses global scope.

But introduces additional concept.

Wart clarifications: Internal "types"

Function *vs* Callable

`typeof f === 'function'` → f is *Callable*

`({}).toString.call(f) === '[object Function]'`
→ f is *a Function*

Non-functions may not claim to be functions

Non-arrays, non-dates, etc… Implied invariants

Non-array `arguments` inherits from `Array.prototype`.

Dates try ISO format first

Strict mode: Support Defensive Code

```
pt.x = 'foo'; // fails how?
```

In ES3 or ES5-nonstrict: Failure is silent.
Execution proceeds assuming success.
Defense impractical. Can't check after every assignment to see if it succeeded.

In ES5-strict code: Failed assignments throw.

Strict mode:
Per script opt-in

```
<script>
  “use strict”;
  //… strict program
</script>
```

Strict mode:
Per script or per function opt-in

```
<script>
  “use strict”;
  //… strict program
</script>
…
<script>
  //… nonstrict program
  function foo() {
    “use strict”;
    //… body of strict function foo
  }
</script>
```

Strict mode:
Per script or per function opt-in

```
<script>
  “use strict”;
  //… strict program
</script>
…
<script>
  //… nonstrict program
  function foo() {
    “use strict”;
    //… body of strict function foo
  }
</script>
```

Still parses on ES3 as a noop.

Nonstrict global object leakage

```
function Point(x,y) {
  this.x = +x;
  this.y = +y;
}
var pt = Point(3,5); //Oops. Forgot “new”
```

Nonstrict: `{window.x = 3; window.y = 5;}`

Strict: `{undefined.x = 3; undefined.y = 5;}`

Safer. And happens to throw quickly in this case

Nonstrict this-coercion hazards

```
Boolean.prototype.not = function() {
    return !this;
};

        true.not()  // ?
        false.not() // ?
```

Nonstrict this-coercion hazards

```
Boolean.prototype.not = function() {
    return !this;
};
```

Nonstrict:
```
true.not()  // false
false.not() // false
```

Strict:
```
true.not()  // false
false.not() // true
```

ES5 functions encapsulated?

```
function foo(x,y){bar();}
```

```
function bar() {
  foo.arguments[0] = ‘gotcha’;
}
```

ES5 functions encapsulated?

```
function foo(x,y){bar();}
```

Nonstrict: `arguments` joined to parameters.

de facto: `foo.caller, foo.arguments, arguments.caller`

de jure: `arguments.callee`

```
function bar() {
  foo.arguments[0] = 'gotcha';
}
```

ES5-strict functions encapsulated

```
function foo(x,y){bar();}
```

Nonstrict: `arguments` joined to parameters.

de facto: `foo.caller, foo.arguments, arguments.caller`

de jure: `arguments.callee`

Strict: `arguments` not joined

defused: `foo.caller, foo.arguments, arguments.caller`

defused: `arguments.callee`

All throw on access, to help catch porting bugs.

Strict functions safe even from non-strict code.

ES5-nonstrict isn't statically scoped

```
function foo() {
    var xfoo = 4; …
    xFoo = 8;       //Misspelling makes global var
}
with (o) {…}        //Attractive idea badly botched
delete foo;         //Dynamically delete static var
eval('var x=8'); x //Dynamically add static var
```

All are fixed or rejected by ES5-strict.

ES5-strict is statically scoped, but…

```
function sqFnList(a) {
   var b = [];
   for (var i = 0; i < a.length; i++) {
     var sq = a[i] * a[i];
     b[i] = function() { return sq; };
   }
   return b;
}

sqFnList([3,4,5])[0]() // ?
```

But static scoping isn't lexical scoping

```
function sqFnList(a) {
    var b = [];
    for (var i = 0; i < a.length; i++) {
      var sq = a[i] * a[i];
      b[i] = function() { return sq; };
    }
    return b;
}

sqFnList([3,4,5])[0]() //Returns 25. Why?
```

But static scoping isn't lexical scoping

```
function sqFnList(a) {
    var b = [];
    for (var i = 0; i < a.length; i++) {
      var sq = a[i] * a[i];
      b[i] = function() { return sq; };
    }
    return b;
}

sqFnList([3,4,5])[0]() //Returns 25. Why?
```

Because of **var** hoisting.

But static scoping isn't lexical scoping

```
function sqFnList(a) {
   var b = []; var sq;
   for (var i = 0; i < a.length; i++) {
     sq = a[i] * a[i];
     b[i] = function() { return sq; };
   }
   return b;
}

sqFnList([3,4,5])[0]() //Returns 25. Why?
```

Because of **var** hoisting.

Lexical Scoping just missed ES5 train. How to cope?

```
function sqFnList(a) {
   var b = []; var sq;
   for (var i = 0; i < a.length; i++) {
     sq = a[i] * a[i];
     b[i] = function() { return sq; };
   }
   return b;
}

sqFnList([3,4,5])[0]() //Returns 25. Why?
```

Because of **var** hoisting.

Higher order array methods

```
function sqFnList(a) {
  return a.map(function(ai) {
    var sq = ai * ai;
    return function() { return sq; }
  });
}

sqFnList([3,4,5])[0]() //Returns 9. Whew.
```

Closure-based iteration helps avoid var hoisting hazards

Higher order array methods

```
anArray.forEach(fn(e,i))
   .map(f(e,i)→r) → array of rs
   .every(f(e,i)→boolean) → boolean // ∀
   .some(f(e,i)→boolean) → boolean // ∃
   .filter(f(e,i)→boolean) → subset
   .reduce(f(r,e,i)→r, r?) → r
   .reduceRight(f(r,e,i)→r, r?) → r
```

JSON: S-expressions of the web

XML supposed to be S-expressions of the web.

JSON: S-expression simplicity regained

- From instigator of ES3.1 simplification.
- Cross origin requests (XHR2,XDR) coming.
- How to obtain remote data?

Today: Can parse quickly or safely, choose one

- JSONP → script tag → full vulnerability
- json2.js → regex guarded eval, thin ice
- full parser in JavaScript code → safe but slow
- json_sans_eval → best compromise today

JSON in ES5

```
JSON.stringify(value, replacer?) → string
JSON.parse(string, reviver?) → value
```

Structured (un)serialization customization

The replacer/reviver idea from Java Serialization Streams.
Immune to textual quoting confusions.

Should be in Ecma or w3c DOM standard?

EcmaScript. Also needed server side.

Remaining hazard: Avoid \u2028, \u2029

Closures don’t capture “this”

```
Foo.prototype.meth = function() {
    this.x = 3;
    this.y = 4;
}
```

Closures don’t capture “this”

```
Foo.prototype.meth = function() {
    this.x = 3;
    setTimeout(function(){this.y = 4;},
               2000);
}
```

Function.prototype.bind() helps

```
Foo.prototype.meth = function() {
    this.x = 3;
    setTimeout(function(){this.y = 4;},
               2000);
} //Oops.

Foo.prototype.meth = function() {
    this.x = 3;
    setTimeout(
      (function(){this.y = 4;}).bind(this),
      2000);
} //Whew.
```

Imperfect Lambda Abstraction in ES5

Tennet Correspondence Principle (TCP)

Litmus test of lambda abstraction

Blocks: `{…}` ≡ `(function(){…})();`

but for `return`, `break`, `continue`, **`this`**, `arguments`, `var`, `function`

Expressions: `(…)` ≡ `(function(){return …;})()`

but for **`this`**, `arguments`

Non-TCP `arguments`, others don't hurt much.

Non-TCP **`this`** is a common trap

repaired by "`.bind(this)`".

Remaining fixable ES5 warts

Static, not Lexical Scoping:

- `const`, `let`, nested named `function` declarations.
- Ambient global at bottom of scope chain.

Spec wart: Multiple global objects?

Regular Expressions process UTF-16 code units.

- Non-BMP, Combining chars → breaking change

i18n → insufficient interest?

Decimal? Mozilla to test the waters.

- Extensible numerics?

Conclusions

ES3 decent for scripting in the small
- Hostile to robust serious programs.

Too many odd corners between
- what beginners learn
- what they need to know.

ES5-strict easier language to teach
- Better for scripting in the small.
- Much better for real programs. Actually good.

Further Reading

Final draft standard “EcmaScript Fifth Edition”

Pratap’s “JScript Deviations”

wiki.ecmascript.org

mail.mozilla.org/listinfo/es-discuss