ES4 Minimalist Proposal - Draft

1 Introduction

The ECMA TC39 TG1 working group is revising ES3 based on 3 data points **(1)** fix problems real JS programmers run into every day **(2)** support programming in the large **(3)** design for the next 10 years.

Here is a quick list of features set to be added in ES4:
Naming / scoping constructs added - classes, interfaces, packages, namespaces (these include ways to declare them as well as new syntax to use them).
Strong typing added; this includes int, uint, decimal and other primitive types.
New OOP-related constructs (both definition and use): bound methods, user-defined operators, etc.
Structural types
Generics
'const', 'static' and 'prototype' modifiers on variables.
The full gamut of modifiers on functions: static, native, override.
Same for classes: including a 'dynamic' modifier.
private/internal/protected/public access attributes.
Class constructors.
"to" and "cast" operators.
Destructuring assignment
Iterators and Generators
Let binding
Proper tail calls
Getters and Setters
Catchalls

It is possible to provide a justification for each of these features. At the same time, though, the fact that a justification is available for every feature is in itself not very convincing.

One approach to solving the problems JS programmers face is to make the language large and complicated. We might end up with multiple ES4 implementations with their own set of incompatibilities (and bugs), making it difficult to treat the web as a stable platform. An alternative approach would be to look at a few key "holes" in ES3 and fill them. This is the "minimalist" approach.

This document approaches the problem from a minimalist perspective, and suggests changes that will add value but at the same time not destabilize the web.

Note:
(1) This is just a draft.
(2) Each of the features mentioned in this document can be used as a jump-off point for other features that the ECMA committee wants to see in ES4. I have not called those out, though.
(3) I have avoided talking about security, and features like language support for debuggability, etc.

1.1 Goals

Improve compatibility among various ECMAScript implementations (especially browser-based) by:

- Rewriting the specification for rigour and clarity
- Adding commonly implemented and used extensions to the standard
- Additional incremental extensions to ES3 supporting current usage experience and best practices
- Minor language changes to correct well known performance or reliability issues

2 A minimalist recommendation for ES4

2.1 Rigourous standards document

The ES3 specification lacks rigour. It is difficult to read and very difficult to understand. We need to fix that with ES4. The required rigour can be in the form of rigourous prose, reference implementation, etc. Our team owns the JScript and JScript.NET implementations, and so as an implementor we would immediately benefit (as would all implementors).

2.2 Targeted additions to Array/String

Frameworks would benefit from such conveniences without too much additional complexity in engine implementations. Frameworks like Atlas, Live, YUI, Prototype, Dojo, etc. routinely augment the built-in types. There are various reasons for this – (a) the built-ins are deficient and lack the "convenience" methods that have evolved out of AJAX (b) these Frameworks require specialized methods on the built-in objects. Of these (a) contributes to unnecessary duplication and code bloat. The convenience methods to add would be as follows.

2.2.1 Array additions

```
every, filter, forEach, indexOf, lastIndexOf, map, some
reduce, reduceRight, setSlice
```

2.2.2 String additions

```
substring, charAt, charCodeAt, indexOf, lastIndexOf,
toLowerCase, toUpperCase, toLocaleLowerCase, toLocaleUpperCase, localeCompare,
match, search, replace, split, substr, concat, slice,
trim
```

Many of these are available in JS1.7, and also proposed in ES4.

2.3 JSON support

JSON is a simple lightweight serialization format that is much used by Frameworks. Atlas, Live, YUI (it was introduced by Yahoo) are heavy users. Introduce safe JSON support as proposed in ES4 under "JSON encoding and decoding".

2.4 Richer reflection capability on function and closures

It is not possible to determine the name of a function or the names of its arguments without parsing the function's source – this is a hole in the reflection functionality available through ES3. Functions should have a "name" property that returns the name of the function as a string. The "name" of anonymous functions can be the empty string.

Functions should also have an "arguments" array, containing the names of the arguments. So, for the example `function foo(bar, baz) {…}`, `foo.name` is `"foo"` and `foo.arguments` is `["bar", "baz"]`.

Similar reflection capability must be made available on closures.

2.5 arguments array as "Array"

Make the arguments array an "Array". That will enable consumers to iterate over its properties using the for .. in loop.

2.6 Readonly built-in object contructors

ES3 allows user script code to override built-in object constructors. However, this is not a common use case, and causes more problems than in solves. This should be disallowed in ES4.

2.7 *Property getters/setters support*

Property getters/setters are now so widespread in use that they merit language support. JS1.7 syntax for getters/setters that is quite acceptable.

2.8 *Fine grained control over object properties*

In ES3 it is legal to add, modify, and delete properties on the `Math` object, the `String` prototype and the other built-in objects. While it is a useful, we must also provide ways to prevent that from happening inadvertently.

2.8.1 sealed

Allow creating "sealed" objects; members cannot be added/deleted on such objects.

2.8.2 readonly

Allow creating "readonly" properties. It should be an error to write to a readonly property.

2.9 *Variable declaration*

In ES3 it is legal to use a variable without declaring it. In those situations, the engine automatically creates a new "global" variable. While convenient, this makes programs run slower. By forcing programmers to declare variables not only can we generate faster code, but the compiler can now catch spelling errors in variable names.

2.10Deprecate "with" statement

The 'with' statement allow a "shorthand" form for writing recurring access to objects. Instead of writing:

```
a.b.val = true;
a.b.val2 = true;
```

one can write:

```
with (a.b) {
  val = true;
  val2 = true;
}
```

This looks concise but what happens in the following case:

```
function foo() {
  var val;

  with (a.b) {
    val = true;
  }
}
```

Is that a reference to the local `val` or to `a.b.val`? Because of the potential of expando properties the compiler cannot tell. And just by looking at this piece of code, nor can the programmer tell. The compiler has to generate a by-name lookup of the entire scope chain - this is expensive.

Additionally, there is no safe way to add a property, say `cost`, to `a.b`; every user of the object who put it in a `with` block and who has a local or global variable called `cost` used in that `with` block is going to suddenly break. Every time we add/remove a property, we risk changing the variable lookup semantics of an existing program. Similarly, every time we use a `with` block, we risk accidentally binding to the wrong thing when someone changes the object properties.

If we can't read a program and be confident that we know what it is going to do, we can't have confidence that it is going to work correctly. `with` must be deprecated.

2.11Deprecate semicolon insertion

"Semicolon insertion" refers to the ability to write programs while omitting semicolons between statements. ES3 does two kinds of semicolon insertion:

Grammatical Semicolon Insertion
Semicolons before a closing } and the end of the program are optional.

Line-Break Semicolon Insertion
If the first through the nth tokens of an ECMAScript program form are grammatically valid but the first through the n+1st tokens are not and there is a line break between the nth tokens and the n+1st tokens, the parser tries to parse the program again after inserting a semicolon token between the nth and the n+1st tokens.

Grammatical semicolon insertion is implemented by the syntactic grammar's productions, which simply do not require a semicolon in the cases called out. Line breaks in the source code are not relevant to grammatical semicolon insertion.

Line-break semicolon insertion cannot be easily implemented in the syntactic grammar. This kind of semicolon insertion turns a syntactically incorrect program into a correct program and relies on line breaks in the source code.

Grammatical semicolon insertion is relatively harmless. On the other hand, line-break semicolon insertion suffers from the following problems:
Line breaks are relevant in the program's source code
The consequences of this kind of semicolon insertion appear inconsistent to programmers

Programmers are confused when they discover that the program

```
a = b + c
(d + e).print()
```

doesn't do what they expect:

That is because the program is parsed as:

```
a = b + c(d + e).print();
```

It is not reasonable for the language definition to require compilers to do semicolon insertion.

2.12Deprecate typeof

The **typeof** operator returns a string based on the type of its operand. It does not adequately differentiate between its operands. For e.g. `typeof(Array())` is considered to be `object`. While the `typeof` operator cannot be fixed now, it can be deprecated and an alternate set of operators can be provided as follows:

```
isObject
isArray
isFunction
isString
isNumber
isBoolean
isNull
isNaN
isUndefined
```

These will return 'true' if their operand is of the desired type.

2.13let bindings to give true block scoping

JS programmers use the `(function() { /* body */ })()` idiom to fake block scope. i.e. by creating, and immediately applying, an anonymous function. However, this idiom has several drawbacks:
(1) the meaning of “return” is changed. A return statement would merely return from the function application expression, and not from the surrounding function.
(2) the meaning of “arguments” is changed.

We need a well behaved local scoping construct for variables and functions. In ES4, "let" binding (aka block expressions) introduces such a construct. `let` can be used to declare variables scoped to a block, or to a particular expression.

3 Conclusion

ES3 has seen resurgence. And we must protect that! ECMAScript went from prototype to world wide acceptance in a remarkably short time. This led to premature standardization, locking some mistakes and inconveniences into the language specification. Many of these mistakes can be fixed without sacrificing compatibility. Doing that would make an already fine language even better, and in the long term better serve the web community.

4 Revision history

Pratap Lakshman, 1 March 2007.