# Compatibility Between ES3 and Proposed ES4[1]

Revision 1, 29 November 2007

This note describes the level of compatibility between ECMAScript 3rd Edition ("ES3") and Proposed ECMAScript 4th Edition ("ES4").

All known incompatibilities between the two editions are divided into groups: true incompatibilities; new names introduced by ES4 (as properties and global names); and behavior that is implementation-dependent or unclear in ES3 but is mandated or clarified in ES4. Of the twelve true incompatibilities listed below, at most two—read-only top-level names and changes to the computation of `this` in certain calls—have not already been implemented in one or more ES3 implementations[2] in widespread use on the world-wide web, where the compatibility requirements for the language are likely the most stringent. Of the other, non-true incompatibilities, most are implemented in one or more of those ES3 implementations.

Additionally, ES4 incorporates some behavior strictly to be backward compatible with ES3, and avoids making some obvious bug fixes for the same reason. Both groups are outlined later in this paper.


**1. True Incompatibilities** **2**
- 1.1. Evaluation of a *RegExp* Literal Results in a new *RegExp* 2
- 1.2. *RegExp.prototype* is a *RegExp* 2
- 1.3. The Value of *length* on Function Objects 3
- 1.4. Top-level Names are Read-Only, not Meta-Programming Hooks 3
- 1.5. Language Constructs Use the Global *Object* and *Array* Classes 4
- 1.6. Bindings for *catch* Variables and Named *function* Expressions 5
- 1.7. Computation of *this* in Calls to Lexically Nested Functions 6
- 1.8. New Keywords 6
- 1.9. Order of Conversion Operations in some Comparisons 7
- 1.10. No Stripping of Unicode "class Cf" Characters 7
- 1.11. The DontDelete Status of a Variable Cannot be Changed by *eval* 8
- 1.12. *Canonicalize* Does not Call *String.prototype.toUpperCase* 8

**2. New Global Names** **9**

**3. New Property Names** **9**
- 3.1. Prototype Methods 9
- 3.2. Value Properties 10
- 3.3. Class Methods 10
- 3.4. The *arguments* Object 11
- 3.5. String Indexing 11

**4. Implementation Dependencies** **12**
- 4.1. *Date.parse* Must Give Precedence to ISO Dates 12
- 4.2. Property Enumeration Order 12

**5. Compatibility-Preserving Decisions** **12**
- 5.1. Automatic Wrapping of "Primitive" Values in Some Contexts 12
- 5.2. Bug Fixes That Were Not Made 13


[1] Based on the 2007-10-23 Proposed ECMAScript 4th Edition Language Overview, the (unpublished) 2007-09-20 Library Specification draft, and the ES4 wiki and bug tracking system. See http://www.ecmascript.org/.

[2] The ES3 implementations are Internet Explorer 6, Firefox 2.0, Opera 9.20, and Safari 3 (beta), all on Windows XP SP2.

# 1. True Incompatibilities

A *true incompatibility between ES3 and ES4* is defined as:

> Any change to ECMAScript introduced in ES4 where code written for a conforming ES3 implementation would generate a different answer in a conforming ES4 implementation, provided the answer is not dependent on implementation-dependent or illegal (exception-throwing) behavior in ES3.

Not included in the definition of true incompatibility are language extensions that are explicitly allowed by Chapter 16 of the 3rd Edition Specification (E262-3) such as the appearance of new global names, new properties on predefined objects or their prototypes, or compatible syntax extensions.

## 1.1. Evaluation of a *RegExp* Literal Results in a new *RegExp*

***Description***

In ES3 a regular expression literal like `/a*b/mg` denotes a single unique `RegExp` object that is created the first time the literal is encountered during evaluation. In ES4 a new `RegExp` object is created every time the literal is encountered during evaluation.

***Rationale***

This is a bug fix. The create-once behavior in ES3 is contrary to programmer expectation.[3] If a `RegExp` object created by a literal is used for matching, its `lastIndex` property will be left in a non-initial state after a match; programmers who expect `lastIndex` to be reset to zero the next time the literal is evaluated (because they expect a new `RegExp` object to be created) will be surprised. The bug bites even advanced programmers.

***Impact***

The changed behavior is observable because `RegExp` objects are mutable and the literal no longer corresponds to a single, unique object. In practice the change is most observable in that the `lastIndex` property of the (new) `RegExp` object is in an initial state every time the literal is evaluated, as desired.

***Implementation precedent***

Internet Explorer 6 and Safari 3 create a new `RegExp` object every time the literal is encountered during evaluation.

## 1.2. *RegExp.prototype* is a *RegExp*

***Description***

In both ES3 and ES4 the prototype object for a constructor is of the same type as the objects returned by the constructor. For example, `Array.prototype` is of type `Array`. ES3, however, makes an exception for the `RegExp` type: `RegExp.prototype` is a plain `Object`. ES4 removes the exception.

***Rationale***

This is a bug fix. The restriction in ES3 has no clear motivation, and it is not entirely consistent. For example, `RegExp.prototype.toString` requires its `this` object to be a `RegExp` object, but because `RegExp.prototype` is itself not a `RegExp`, the expression `String(RegExp.prototype)` will always throw a `TypeError` (unless the `toString` method is replaced by the user program).

***Impact***

Programs can observe that the class name of `RegExp.prototype` has changed.

[3] Evidence of this may be found at https://bugzilla.mozilla.org/show_bug.cgi?id=98409.

Programs that transfer the `RegExp` prototype to other constructor functions will find that objects constructed by those functions have additional properties (in their prototype).

***Implementation precedent***
Internet Explorer 6, Firefox 2, Opera 9.20, and Safari 3 all have a `RegExp.prototype` object that is itself a `RegExp`.

## 1.3. The Value of *length* on Function Objects

***Description***
In ES3 some predefined function objects have `length` properties whose values deviate from the standard rule, which requires that the length property represent the number of formal parameters in the function definition (or in ES4 terms, the number of fixed and optional parameters, not counting `this` type-bounds or rest arguments). The standard rule is followed for all function objects in ES4. As a consequence, the following predefined function objects have new length values:

| *Function* | *ES4 value* | *ES3 value* |
|---|---|---|
| `Function` | 0 | 1 |
| `Array` | 0 | 1 |
| `Math.max` | 0 | 2 |
| `Math.min` | 0 | 2 |
| `Array.concat` | 0 | 1 |
| `Array.push` | 0 | 1 |
| `Array.unshift` | 0 | 1 |
| `String.fromCharCode` | 0 | 1 |
| `String.prototype.concat` | 0 | 1 |
| `String.prototype.indexOf` | 2 | 1 |
| `String.prototype.lastIndexOf` | 2 | 1 |

***Rationale***
This is language cleanup. The irregularities in ES3 are not clearly motivated by utility. Furthermore, since ES4 allows the formal parameter lists of predefined functions to be expressed with greater clarity (using optional and rest arguments) than does ES3, following the common rule in ES4 makes more sense. Eliminating special cases is good for the language in general and also benefits language implementations.

***Impact***
The change is visible to programs that inspect or act on the values of the `length` properties of functions.

***Implementation precedent***
Firefox 2 has `String.prototype.concat.length` = 0. Internet Explorer 6 has `String.fromCharCode.length` = 0, `String.prototype.indexOf.length` = 2, and `String.prototype.lastIndexOf.length` = 2. Opera 9 is known to deviate from ES3 in the value of `length` properties on predefined methods not listed in the table.

## 1.4. Top-level Names are Read-Only, not Meta-Programming Hooks

***Description***
The following top-level bindings were writable in ES3 but are read-only in ES4.

- The primitive values `undefined`, `NaN`, and `Infinity`
- The following names that were constructor functions in ES3 and are classes in ES4: `Object`, `Function`, `Array`, `String`, `Boolean`, `Number`, `Date`, `RegExp`, `Error`, `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, and `URIError`

There are metaprogramming and aspect-oriented programming use cases for allowing the predefined constructors in ES3 to be changed, and the ES3 specification incorporates language that requires the implementation to use the current value of `Object` (E262-3 11.1.6, 12.14, 13), `Array` (E262-3 11.1.4,

15.4.4.4, 15.4.4.10, 15.4.4.12, 15.5.4.14), or `Date` (E262-3 15.9.2.1). The user program can override the `Object` constructor, say, to add extra properties to every new `Object` instance without the cooperation of the user program.

***Rationale***
This is a bug fix. It is silly to be able to redefine `undefined`, `NaN`, and `Infinity`—programmers want to depend on these values. (Programmers write the more obscure `(void 0)`, `(0/0)`, and `(1/0)` to avoid having to depend on these names).

The mutability of constructors in ES3 has three problems. One problem is that the user-defined constructors are not always called. When the implementation creates new objects of some types E262-3 sometimes mandates the use of the original constructor and prototype values instead of the program overrides, and inconsistent data may result.

The second problem is that the changed constructors are called when the program might wish them not to be. When the implementation evaluates object and array initializer expressions—`{ x: 10 }` and `[10, 20]`—it is supposed to create the new objects as if by "`new Object`" and "`new Array`". Scripts cannot always guarantee that data will not be stolen.[4] (The next section describes a related problem.)

Finally, the predefined constructors in ES4—in the form of classes—are used as type names in annotations on variables and parameters, as well as in type dispatched control structures, and allowing the predefined bindings to be changed would make ES4 code unpredictable at best.

***Impact***
The change will be visible to programs that try to update these bindings, in two ways: first, the values are not updated (though no exception is thrown); second, metaprogramming techniques that rely on being able to change the bindings will no longer work.

***Implementation precedent***
No ES3 implementations are known to make these top-level names read-only, and the impact of that change is therefore unknown. However, ES3 implementations do not always respect E262-3 with regard to invoking user-defined constructors, so there is precedent for blocking the metaprogramming techniques. The next section describes two important cases in more detail.

## 1.5. Language Constructs Use the Global *Object* and *Array* Classes

***Description***
ES3 requires object and array initializers, `catch` clauses, and named `function` expressions to construct new objects "as if by `new Object`" or "as if by `new Array`". The meaning, if not the intent, of the requirement is that the appropriate `new` expression is evaluated in the scope of the language construct occurrence, with the consequence that any non-global binding of the names `Object` and `Array` will be used by the expression.

ES4 requires the global, standard bindings of `Object` and `Array` to be used to construct new objects for object and array initializers. As described in the next section, ES4 also requires that `catch` clauses and named `function` expressions not allocate objects in the way required by E262-3.

***Rationale***
This is a bug fix. The purpose of the ES3 requirement is to allow the user program to redefine the `Object` and `Array` constructors globally and have literals create instances of the new types, not to allow the user program to pick up arbitrary bindings for those names.

[4] See http://getahead.org/blog/joe/2007/03/05/json_is_not_as_safe_as_people_think_it_is.html for a broader study.

***Impact***
Programs that depend on being able to pick up shadowing bindings may experience changed behavior. Additionally, as described in the previous section, the global `Object` and `Array` are read-only in ES4 and cannot be redefined. Object and array initializers will always create instances of the predefined classes.

***Implementation precedent***
Internet Explorer 6, Opera 9.20, and Safari 3 do not respect either local or global rebindings of `Object` and `Array`, but use the original `Object` and `Array` constructors.

## 1.6. Bindings for *catch* Variables and Named *function* Expressions

***Description***
ES3 specifies that the variable name introduced by a `catch` clause and the function name introduced by a named `function` expression should both be bound by creating a new `Object` instance which is pushed on the scope chain and in which a property of the name is created to hold the value. Because the names are bound by objects that are not activation objects, those objects become the value of `this` in the invocation should the name be called as a function (E262-3 11.2.3), and the binding objects will therefore become visible to the program. If the program adds properties to a binding object, those will become visible to the program because of the `with`-like scoping mechanism used to introduce that object.

An example:

```
function f() {
    this.x = 10
}

function g() {
    var x = 20
    try {
         throw f
    }
    catch (exn) {
        exn()
        print(x)             // prints 10, not 20
    }
}

g()
```

ES4 specifies that these names should both be bound as if by `let`, that is, by a block scope object. Block scope objects are treated like activation objects, and `this` will either be unchanged or the global object in the invocation. (See also the next section, “this”).

***Rationale***
This is a bug fix. The ES3 behavior is most likely the consequence of reusing the `with` mechanism for `let`-like bindings without fully considering how that would make the private scope objects visible to the program as part of the function calling machinery.

There is also a security hole lurking in the ES3 behavior. The specification requires the binding object to be created “as if by ‘`new Object`’”, so the problem with mutable top-level bindings described in the previous section applies here as well.

***Impact***
The change is visible to programs that invoke the values of those bound names as functions and then access the `this` value, which will never be the binding object in ES4.

***Implementation precedent***
Firefox 2 and Internet Explorer 6 implement `let`-like scoping for catch clauses. Opera 9.20 implements `let`-like scoping for named function expressions.

## 1.7. Computation of *this* in Calls to Lexically Nested Functions

***Description***
ES4 specifies that when a function that is defined by a *FunctionDefinition* lexically nested inside some other function is called directly by naming it in the call, then the caller's value of `this` is transferred to the callee. (The called function does not need to be lexically nested inside the calling function, just inside some function—the called function could be a function containing the caller, or a sibling of such a function.) ES3 specifies that the value of `this` would be the global object in that case.

For example, this program prints "`local`" in ES4 but "`global`" in ES3:

```
var x = "global"
var o = { m: function () {
                function g() { print(this.x) }
                g()
            },
          x: "local"
        }
o.m()
```

***Rationale***
This is a bug fix. Reverting the value of `this` to the global object is not useful behavior, and creates a hazard where global variables may be set or created unintentionally. The changed behavior especially supports ES3-style method functions that abstract common functionality as functions, or that use functional abstraction to clarify the code.

***Impact***
Code that attempts to get at the global object by invoking a function that just returns `this` can no longer do that under the circumstances described above.

***Implementation precedent***
Unknown.

## 1.8. New Keywords

***Description***
ES4 defines as keywords some identifiers that are valid names in ES3. Most of these keywords are only reserved in some contexts, like `override`, but others are reserved everywhere, like `class`, `let`, and `yield`. Some of the keywords are *future reserved words* in ES3 and should not occur in ES3 programs, but in practice ES3 implementations allow them to be used as names and programs do.

The consequence of the larger set of keywords is that there are some ES3 programs that are not valid ES4 programs, because they use the ES4 keywords as names in contexts where ES4 does not allow them.

(Note here that `eval(s)` does not suffer from a compatibility problem because it uses the ES3 keyword set unless an extra parameter is passed asking for the ES4 keyword set.)

***Rationale***
This is a consequence of language evolution. The committee felt it was necessary to introduce new syntactic facilities into ES4 that required the new keywords.

Every evolving programming language struggles with this problem; in ES4 it has been sought to minimize the problem by making most new keywords contextual, and allowing any keyword to be used in some contexts such as property names.

***Impact***
ES4 implementations that accept arbitrary ES3 content must make it possible for the provider of the content to specify the dialect; the provider must on the other hand specify the dialect (typically by a MIME type).

***Implementation precedent***
Firefox 2 supports the `let` and `yield` keywords if the script that uses them is loaded with the MIME type "`application/javascript;version=1.7`".

## 1.9. Order of Conversion Operations in some Comparisons

***Description***
ES3 specifies the binary comparison operators (<, >, <=, >=) in terms of an abstract comparator algorithm (E262-3 11.8.5) that is called by the (semantic) functions that implement the comparison operators. A semantic function passes the operator's two operands as arguments when calling the comparator algorithm. The comparator algorithm subsequently invokes the internal operator `ToPrimitive` on the two arguments in the order they are received. The calls to `ToPrimitive` are significant in that they may have user-observable effects because they may invoke user-defined `valueOf` or `toString` methods on the operand. Unfortunately, the semantic functions that call the comparator algorithm sometimes pass the operator's left-hand-side argument followed by its right-hand-side argument, and sometimes the reverse. Thus the program may observe that (`a > b`) has right-to-left order of evaluation as observed by the sequence of methods calls.

ES4 specifies that the left-hand-side argument to the comparison operator is converted to primitive before the right-hand-side argument is.

***Rationale***
This is a bug fix. Elsewhere, ES3 is careful about maintaining left-to-right order of evaluation, and the behavior for the comparison operators is a language bug.

***Impact***
Programs that depend on right-to-left order of primitive conversion for > and <= may experience changes in behavior.

***Implementation precendent***
Internet Explorer 6, Firefox 2, and Opera 9.20 all perform primitive conversions in left-to-right order.

## 1.10. No Stripping of Unicode "class Cf" Characters

***Description***
ES3 specifies that Unicode format control characters ("class Cf") must be stripped from all source text before it is processed (E262-3 7.1). In practice, not all ES3 implementations do this, and those that do report that they receive complaints about it (because programmers find it convenient to embed these characters in string and regular expression literals).[5]

ES4 therefore specifies that class Cf characters should not be stripped from source text at all, with the exception of byte order marks (BOMs), which must continue to be stripped everywhere.

***Rationale***
This is for user convenience. The expectation is that the program text should contain the same characters that the programmer has written into it. BOMs are the exception to that rule. BOMs have meaning only at the beginning of a text, and their appearance elsewhere in a text is usually a result of cutting and pasting

[5] See for example https://bugzilla.mozilla.org/show_bug.cgi?id=274152.

Unicode text with tools that are not fully Unicode-aware. Such misplaced BOMs are prevalent in practice and ES4 therefore makes special allowance for them.

***Impact***
Programs that contain class Cf characters may stop functioning (syntax errors will be reported) because class Cf characters are not generally allowed by the ECMAScript grammar except in string and regular expression literals and comments.

***Implementation precedent***
Internet Explorer 6 and 7, Firefox 2, and Opera 9.20 do not strip class Cf characters from any program text, with possible variations in how BOMs are handled. (Results were obtained earlier by Ecma TC39-TG1 members; Safari was not examined.) Representatives of Mozilla and Opera both report that BOM stripping is important in practice.

## 1.11. The DontDelete Status of a Variable Cannot be Changed by *eval*

***Description***
A binding introduced by the `var` statement has its DontDelete attribute set, but two rules of ES3 together create a situation where it becomes possible to remove a variable from a scope: A binding introduced by a `function` definition replaces any `var` binding in the same scope; and the `eval` function can be used to introduce a `function` binding in a scope. The following program first introduces a variable "`x`" and then manages to remove it:

```
var x
eval("function x() {}")
delete x
```

ES4 specifies that `eval` is not allowed to change the DontDelete attribute of a binding.

***Rationale***
This is a bug fix. The ability to remove declared names from a scope makes program analysis and optimization harder or impossible, for no obvious gain.

***Impact***
Programs that use this trick to delete variables may change behavior.

***Implementation precedent***
Internet Explorer 6, Firefox 2.0, Opera 6.20, and Safari 3 all preserve the DontDelete attribute as required by ES4 in function scopes; Opera and Safari preserve it also in the global scope.

## 1.12. *Canonicalize* Does not Call *String.prototype.toUpperCase*

***Description***
ES3 defines a helper function called `Canonicalize` that is used during case-insensitive regular expression matching to perform case conversion (E262-3 15.10.2.8). The definition of `Canonicalize` says that it converts a character to upper case "as if calling `String.prototype.toUpperCase`". In other contexts the language "as if ..." is interpreted as a requirement that the implementation actually perform the action described.

ES4 defines an internal helper function for case conversion that is shared between `Canonicalize` and `String.prototype.toUpperCase`, and does not require the latter function to be called from the former.

***Rationale***
This is a bug fix. The language in ES3 turns `String.prototype.toUpperCase` into a hook, but this is probably inadvertent and a consequence of the specification writer not wanting to define an internal helper function for case conversion, or simply not seeing the consequence.

***Impact***
Negligible.

***Implementation precedent***
None of Internet Explorer 6, Firefox 2.0, Opera 9.20, or Safari 3 call the current value of `String.prototype.toUpperCase` to perform case conversion during regular expression matching.

# 2. New Global Names

***Description***
ES4 introduces new, visible global names. This is only a quasi-incompatibility, since E262-3 Chapter 16 explicitly allows new global names to be added by any implementation.

Most new global names (like `hashcode` or `int`) are not visible to ES3 code because they are defined in the `__ES4__` or `intrinsic` namespaces, which are only open in ES4 code.

Still, there are two new names visible even to ES3 code.

- `__ES4__`: a constant namespace binding.
- `__ECMASCRIPT_VERSION__`: a constant integer value describing the latest edition of the language accepted by the implementation.

***Rationale***
This is a consequence of language evolution. The new names make new functionality available to the program.

***Impact***
ES3 programs that use these names for their own variables will find that they can't do so in an ES4 implementation.

***Implementation precedent***
Implementations commonly add new top-level names, though often they are not read-only.

# 3. New Property Names

ES4 introduces new, visible property names on objects. This is a quasi-incompatibility, since E262-3 Chapter 16 explicitly allows new property names to be added by any implementation. Several ES3 implementations already implement some of the changes described in this section.

## 3.1. Prototype Methods

***Description***
These are new methods added to prototype objects, broken down by class. All of these properties are DontEnum.

```
Array.prototype:
    every, filter, forEach, indexOf, lastIndexOf, map, reduce, reduceRight, some

Date:
    toISOString
```

***Rationale***
This change is a consequence of language evolution. The new names make new functionality available to the program.

***Impact***
The new names are visible to "`in`" tests; they are not ReadOnly, so new values can be stored in them.

***Implementation precedent***
Firefox has many new `Array` methods.

## 3.2. Value Properties

***Description***
These are new properties added to instances of `Date` and `RegExp`.

The `Date` properties are getter/setter pairs that delegate to existing getter and setter methods. For example, `time` delegates to the intrinsic `getTime` and `setTime` methods. These properties are DontEnum and DontDelete; the `timezoneOffset` property is ReadOnly.

The new `RegExp` properties, like other properties on `RegExp`, are DontEnum, DontDelete, and ReadOnly.

`Date`:
`time`, `year`, `fullYear`, `month`, `date`, `day`, `hours`, `minutes`, `seconds`, `milliseconds`, `timezoneOffset`, `UTCFullYear`, `UTCMonth`, `UTCDate`, `UTCDay`, `UTCHours`, `UTCMinutes`, `UTCSeconds`, `UTCMilliseconds`

`RegExp`:
`extended`, `sticky`

***Rationale***
The `Date` properties help bring `Date` into line with other predefined classes, whose value properties are generally read and written using property reference and property update syntax, not getter and setter methods.

The `RegExp` change is a consequence of language evolution. The properties reflect the settings of the "`x`" and "`y`" flags in the regular expression.

***Impact***
The new properties are visible to "`in`" and "`hasOwnProperty`" tests. They are all DontDelete, typed, and in some cases ReadOnly, and user programs cannot generally use them for anything other than their intended purpose.

***Implementation precedent***
Unknown.

## 3.3. Class Methods

***Description***
These are new properties added to the global constructors (i.e., they are methods on ES4 class objects), broken down by class. These properties are DontEnum, DontDelete, and ReadOnly.

`Function`:
`apply`, `call`

`Array`:
`concat`, `every`, `filter`, `forEach`, `indexOf`, `join`, `lastIndexOf`, `map`, `pop`, `push`, `reduce`, `reduceRight`, `reverse`, `shift`, `slice`, `some`, `sort`, `splice`, `unshift`

`String`:
`charAt`, `charCodeAt`, `concat`, `indexOf`, `lastIndexOf`, `localeCompare`, `match`, `replace`, `search`, `splice`, `split`, `substring`, `toLowerCase`, `toLocaleLowerCase`, `toUpperCase`, `toLocaleUpperCase`, `trim`

```
Date:
    now
```

***Rationale***
This change is a consequence of language evolution. The new names make new functionality available to the program.

***Impact***
The new names are visible to "`in`" tests; as they are ReadOnly and DontDelete they are effectively controlled by the implementation, and cannot be used by the program for the program's data.

***Implementation precedent***
Firefox has many new class methods on the `Array` object.

## 3.4. The *arguments* Object

***Description***
In ES3 the prototype of the `arguments` object (E262-3 10.1.8) is the original `Object.prototype`. In ES4 the prototype of the `arguments` object is an object very much like `Array.prototype`, the only difference being that the `toString` method behaves like `Object.prototype.toString`.

***Rationale***
This is a bug fix. Providing the `arguments` object with `Array` methods is a much-requested feature.

***Impact***
Programs will find that the `arguments` object has new, named properties; these are visible to the "`in`" operator but not to `for-in` enumeration.

***Implementation precedent***
Opera 9.20 provides `Array` methods on the `arguments` object.

## 3.5. String Indexing

***Description***
ES4 allows the use of bracket notation, that is "`s[x]`", to denote a call to the original intrinsic `charAt` method on `string` and `String` objects provided `x` is inside the range of characters in the string. Outside the range of characters, the behavior is as in ES3: a normal property get or set. Since strings are immutable, property sets inside the range of characters are ignored.

***Rationale***
This new feature is motivated by user convenience. Code that processes strings character by character is tedious to write and hard to read because of all the calls to `charAt`. Without string indexing syntax, ES4 code that wants good performance and reliable semantics would need to call `intrinsic::charAt`, making the code even less readable (other things being equal). The syntactic support for string indexing provides readable code, reliable semantics, and optimization opportunities.

***Impact***
Characters in the range of the string will appear as new properties on the string object, and programs like the following will experience changed behavior:

```
x = new String("abc")
x[1] = 3
x[1]
```

In ES4 the result is `"b"`, not `3`.

Note that instances of `string`, unlike instances of `String`, are fully immutable and do not experience this problem; string literals are in turn `string` instances, so it is only by using the String wrapper class (or placing properties on `string.prototype`) that problems may occur.

***Implementation precedent***
Firefox 2, Opera 9.20, and Safari 3 all provide this functionality.

# 4. Implementation Dependencies

The following is a list of quasi-incompatibilities: behavior mandated by ES4 where ES3 left it implementation-dependent, and where programs depending on particular implementations may experience changes.

## 4.1. *Date.parse* Must Give Precedence to ISO Dates

***Description***
ES4 mandates that date strings that match the ISO-8601 subset date grammar in ES4 must be parsed as ISO dates.

***Rationale***
This is a consequence of language evolution. ES3 mandates only that `Date.parse` must parse the output of `Date.prototype.toString` and `Date.prototype.toUTCString`. ES4 adds the `Date.prototype.toISOString` method. ISO time stamps are given precedence not only because they are rigidly specified and not likely to have been the format produced by the existing two formatting methods in ES3 implementations, but also because ISO time stamps are unambiguous.

***Implementation precedent***
Opera 9.20 parses some ISO date prefix strings, like "`2007`", "`2007-01-01`" and "`2007-01-01T12`" (though it does not allow "`:`" separators following the "`T`"). Other browsers do not parse any of these strings.

## 4.2. Property Enumeration Order

***Description***
ES3 does not define the order of enumeration of properties in a `for-in` loop. ES4 makes this order fixed (it is property insertion order, starting the enumeration with the object that is the operand to the loop and then walking up that object's prototype chain).

***Rationale***
Following the introduction of insertion-order enumeration in Netscape Navigator and then in Internet Explorer, the web has come to depend on it.

***Implementation precedent***
Internet Explorer 6, Firefox 2, and Opera 9.20 implement insertion order enumeration, with some variations. (Safari was not tested.)

# 5. Compatibility-Preserving Decisions

## 5.1. Automatic Wrapping of "Primitive" Values in Some Contexts

***Description***
ES3 primitive values—strings, numbers, booleans—are not true objects, and calling a method on a primitive value in ES3 results in a new *wrapper object* being instantiated for the value and the method being called on the wrapper. The wrapper object becomes the `this` object in the call. Arbitrary methods can be added to the wrapper's prototype, so it is possible to capture the wrapper objects. Programs can thus come to depend on the identity of wrapper objects.

Furthermore, since wrappers are constructed for any property access to an ES3 primitive value, programs can read and write arbitrary properties on the primitives. The accesses have no effects, as the wrappers are constructed, accessed, and then discarded, but the programs are legal.

The ES4 values that correspond to ES3 primitive values are true, immutable objects. Calling a method on an ES4 value does not normally result in the construction of a new wrapper object, and reading and writing arbitrary properties normally result in run-time errors.

However, ES4 hides this incompatible change by means of three facilities.

- The prototype object of an ES4 class (`string`, `int`, `uint`, `double`, `decimal`, or `boolean`) that models an ES3 primitive value is shared with the value's wrapper class (`String`, `Number`, or `Boolean`). Methods added to the wrapper class's prototype also become methods on the primitive value.
- When a method without a bound-`this` type is entered and the `this` value is one of the ES4 classes `string`, `int`, `uint`, `double`, `decimal`, or `boolean`, then the `this` value is wrapped and the wrapper becomes the new `this` value.
- The ES4 classes `string`, `int`, `uint`, `double`, `decimal`, and `boolean` allow arbitrary properties of their instances to be read and written, producing the value defined in the prototype (if any) when a property is read and ignoring the value when a a property is written.

The wrapping of primitives on calls to methods that do not have bound-`this` types might at first glance appear to reintroduce the very problem we were trying to avoid, but this is not so. The predefined `String`, `Number`, and `Boolean` prototype methods in ES4 all have bound-`this` types (the types are `Object`, `AnyNumber`, and `AnyBoolean`, respectively), so no wrapping occurs on calls to predefined methods – this optimization is also performed in some existing ES3 implementations. Furthermore, wrapping can be performed lazily; it does not have to occur until a method references its `this` value.

## 5.2. Bug Fixes That Were Not Made

Whether a language feature is or is not a "bug" is necessarily subjective. Some consider it a bug that the scope of `var` is not block scope but rather the scope of the enclosing function or the program; some consider it a bug that `eval` has access to the scope of its calling function. We will not try to define "language bug" clearly here, but will enumerate some ES3 language features that TG1 discussed changing, but in the end left alone.

***Typeof***

The `typeof` operator provides a crude mechanism for classifying values. Its main virtue is that it works reliably across multiple global objects in web browsers. If a value is an object, `typeof` will classify it as `"object"` regardless of the value's originating global object. (The `instanceof` operator provides more accurate testing but is not reliable when objects can be created in several global objects.)

The `typeof` operator has three problems: it classifies the value **null** as `"object"`; it does not reliably reveal whether an object can be called as a function; and it distinguishes only between primitive values and object values: an array object is classified as `"object"`, which is not very helpful.

TG1 decided that `typeof` is beyond rescue and that changing it is fraught with compatibility problems. Programs are known to depend on **null** being classified as `"object"`, and it is likely that programs assume the same for arrays, for example. Even if that were not the case, it's not clear how to extend `typeof` to provide more general information without going all the way and returning reflected type objects, something that would clearly be backward incompatible.

TG1 has decided to clarify that `typeof` returns `"function"` *only* for objects that are instances of `Function` or `Function`'s subtypes. This is compatible with ES3. The decision is significant for ES4,

however, because objects that are callable without being functions—regular expression objects and other objects with `meta::invoke` methods—will not be classified as `"function"`.[6]

ES4 provides facilities that complement `typeof` and `instanceof`:

- The operator `is` tests whether the value that is its left-hand argument is an instance of the type that is its right-hand argument (or one of that type's subtypes): `x is Array`.
- The `is` operator is implicitly invoked by type annotations on variables, providing convenient run-time (or compile-time) assertions: `var p: Array`.
- The function `reflect::typeOf` returns a type meta-object for its argument. The `isSubtypeOf` method of the meta-object can be used to test whether the type is a subtype of another type: `reflect::typeOf(x).isSubtypeOf(Array)`. Classes, interfaces, and type definitions are all reflected as type meta-objects in ES4.
- There is a predefined structural type `Callable` that matches any type that has a `meta::invoke` method: `/abc/g is Callable` is true, but `typeof /abc/g` is `"object"`.
- Predefined types are "the same" across multiple global objects. The type meta-objects for the class `Array` must be different in different global objects because they are mutable, but the underlying type is the same, and "`x is Array`" is true or false independently of whether the value of `x` was created in the global context in which it is tested.

***Eval***

The `eval` function provides a limited form of dynamic scoping: It can insert new bindings in its caller's innermost scope. As a consequence, functions that call `eval` cannot know whether a reference to a variable that is not defined locally denotes one that `eval` introduces or one that is defined in an outer scope. Furthermore, functions that call `eval` can easily shadow global bindings that the evaluated program will need. Finally, though ES3 allows uses of `eval` other than simple calls to the function by name to be flagged as errors, some implementations allow the `eval` function to be treated as a value and called by other names, and in those implementations, any function call is potentially a call to `eval` unless analysis can prove otherwise.

On the one hand, TG1 observed that the specification of eval in ES3 was too weak in practice: Programs depend on calling `eval` as a method on window objects in web browsers, and more rarely on being able to rename `eval`. On the other hand, the ability of `eval` to introduce names into a scope breaks compile-time type checking.

It is impractical to remove `eval` from the language because programs on the web use it heavily (often unnecessarily).

In the end TG1 decided that ES4 has two forms of `eval`: an operator form, which is obtained by using the identifier "`eval`" as the function expression in a function call, and a function form, which can be invoked as a method on the global object or directly as a function under some other name. The latter form does not have access to its caller's scope.

As a concession to program verifiability, ES4 specifies incompatible behavior for `eval` in strict mode: it is not allowed to insert new bindings into the lexical scope of its caller, unless that scope is the global one.

***With***

The `with` statement provides a limited form of dynamic scoping: It inserts an arbitrary object into the lexical scope chain and performs variable lookup in the object every time a scope chain search reaches the object. Though convenient to use, it tends to slow programs down and makes them hard to analyze for compilers and programmers alike.

[6] Experience in Firefox has shown that it's a bad idea to classify regular expression objects, which are callable as functions, as `"function"`, and that has informed TG1's decision.

TG1 considered several proposals to alleviate the situation, among them to remove `with` from the language and require references to `with`-bound variables to be syntactically distinct. Neither is compatible with existing code and both were rejected for that reason.

In the end, ES4 introduces a new form of `with` known as "reformed `with`". The new form allows the object expression in the `with` head to be annotated with a structural record type that lists all the fields (and their types) that the body of the `with` statement may read from the object. Though superficially similar to the normal `with`, the new form does not provide any kind of dynamic scoping. All fields listed in the annotation must be fixtures in the object, with exactly the types listed in the annotation. The utility of reformed `with` lies in the fact that an ES3 style `with` can be rewritten quickly as a reformed `with`, provided the object has the necessary fixtures. (Wrapped objects are acceptable.)

As a concession to program verifiability, ES4 specifies that only the reformed `with` statement is allowed in strict mode.

***Octal numbers***

TG1 had the option to do something about the situation with octal numbers, which are specified only in a non-normative appendix in E262-3. The normative parts of E262-3 do not allow decimal numeric literals to start with the digit `0` in all contexts; they do allow numeric integer literals starting with `0` to be interpreted as octal numbers in some contexts, but non-normative notes actively discourage that. Finally, the non-normative appendix only provides interpretations for strings of digits starting with `0` but not containing the digits `8` and `9`, yet such strings are common in programs on the web.

In the end TG1 decided to leave octal alone and at most tighten up the non-normative prose to define what happens if digits `8` and `9` are seen.

***Delete***

The operator `delete` is used to delete a property or deletable variable. It returns "`true`" if the property was not found, or if the property was found and deleted – and "`false`" only if the property was found but could not be deleted (because it is DontDelete).

ES3 added exception handling and the natural behavior for ES3 (and ES4) is to throw a catchable exception for non-found properties. Making the change would likely break existing programs, however. (Consider a program that knows it may or may not have created some property on an object. It is faster and less cumbersome for the program to delete the property without testing for its presence than it is to test and then delete.)

***Read-Only Properties***

A write to a constant (ReadOnly) variable does not update the variable, but silently does nothing. JavaScript inventor Brendan Eich stated on the es4-discuss list on 12 November 2007 that JavaScript in Netscape Navigator versions 2 and 3 would signal an uncatchable error instead of doing nothing in that situation. The behavior was changed in later versions of JavaScript, presumably to match the language's generally permissive behavior.

ES3 added exception handling and the natural behavior for ES3 (and ES4) is to throw a catchable exception for writes to constant properties. Making the change might break existing programs, however, and the introduction of new ReadOnly names in ES4 and the const keyword to designate properties as ReadOnly increase the chance of breaking programs. There is also the problem that there is no way in ES3 or ES4 to test whether a property is marked ReadOnly (unlike testing for property presence).

## Change Log

29-Nov-2007 *Revision 1*. New title. Added table of contents. Expanded the treatment of read-only top-level names to include metaprogramming issues. Added the section on always requiring the global definitions of `Object` and `Array` to be used. Noted that FF and IE allow the *DontDelete* status of properties to change in the global scope, but not in other scopes. Added the section on `Canonicalize`. Added the section on compatibility-preserving decisions. Added description of the new `Date` getter/setter properties. Added the Change Log.

05-Nov-2007 *Initial version.*

## Licensing terms, Conditions, and Copyright

The following licensing terms and conditions apply to this Incompatibilities Overview document.

1. This Incompatibilities Overview is made available to all interested persons on the same terms as Ecma makes available its standards and technical reports, as set forth at http://www.ecma-international.org/publications/.

2. All liability and responsibility for any use of this Incompatibilities Overview rests with the user, and not with any of the parties who contribute to, or who own or hold any copyright in, this Incompatibilities Overview.

3. THIS INCOMPATIBILITIES OVERVIEW IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS INCOMPATIBILITIES OVERVIEW, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

End of Terms and Conditions