The Predefined ECMAScript 4 Libraries

Incomplete working draft, 27 November 2007 / lhansen@adobe.com

Abstract

At the time of writing there already exists a rough but mostly complete library specification for ES4. This paper will therefore concentrate solely on requirements, design principles and conventions, rationale, and other aspects of the predefined libraries that are implied by their specification but not explained in much detail in the specification document.

Below is a preliminary outline.

Requirements

The following are the requirements for the predefined libraries.

Compatibility

The ES4 library must generally be entirely compatible with the ES3 library.

Exceptions are made for new top-level properties, new properties on predefined classes and their instances, and compatible extensions to existing methods. New properties are explicitly allowed by Chapter 16 of E262-3. Compatible extensions to existing methods are allowed in ES4 only to the extent the change makes use of provisions in ES3 about implementation-defined behavior. For example, the optional second argument to the `propertyIsEnumerable` method is allowed because ES3 states that passing more than one argument to this method has implementation-defined results; no portable ES3 program will therefore be passing a second argument to the method.

Self-hosting

The specification should use ES4 to the greatest possible extent. This requirement is really a consequence of a requirement on the ES4 design, which is that ES4 should be able to describe its own libraries (on the principle that what is good for the language implementation is good for the language user).

The primary consequence of the requirement is that the library will provide some functionality—typically methods—whose current primary use case is the library itself. For example, the `explode` and `implode` methods on `double` and `decimal` are primarily useful for implementing binary serialization of floating-point number, typically a library responsibility. (Though at present the ES4 library does not provide that functionality on any data type.)

Support the Language

There are classes that are in the library that are direct consequences of the artifacts of the language, like `NamespaceSet`, `AnyNumber`, `EnumerableId`, and `Error` and its subclasses. The library contains definitions for these because they will be useful for the program, not because they are profound, hard to write as ES4 code, or performance sensitive.

Be Useful yet Tractable

The library should provide facilities that many programs need, without attempting to be truly comprehensive; TG1 wants to encourage a library infrastructure for ES4, and has included facilities in ES4 for that purpose. A consequence of the class-based model is that it becomes important to define more methods with the class than might be seen as necessary with the older prototype based model; the library sometimes includes more functionality than strictly necessary for this reason.

The library should provide functionality that some programs need and that could be provided provided by user code, but which stands to benefit significantly from native implementations (at least with current implementation technology). Examples are mathematical functions, floating point number parsing and formatting, string searching, sparse array operations.

Conventions

Namespaces

An overview of the namespaces reserved by the system, and their purpose, intended use, etc. What does it mean for a namespace to be reserved?

- __ES4__, __ES<n>__: for hiding names from programs defined on earlier editions. Specifically, when is __ES4__ opened; what might happen in future editions.
- reflect: for reflective methods
- intrinsic: for early binding
- iterator: for iteration protocol
- meta: for catch-alls, meta-level hooks

Prototype, intrinsic, and static methods

- The purpose of each of those method classes
- Why there is a special "prototype" modifier
- Why there are more static generic methods now than before
- The "canonical" implementation of a method
- How call forwarding works
- Principles for signatures on the different classes
- Interaction with strict mode

Wrappers

- Existence of wrapper classes
- Compatibility concerns with not wrapping

Type conversion

Invariably implemented as a `meta::invoke` method on the class object, which converts a value to that class if possible. For wrapper classes, instances of the primitive classes are returned. Both mirror the way things are done in ES3.

Types and Reflection

- underpinnings of reflect (Ungar/Bracha)
- sharing of built-in types (but not type objects) across global environments

Iterators and generators

- underpinnings of iterators
- rationale for shallow generators

Magic

- Shared prototypes for some types
- Internal constructors (not expressible in ES4) for some types
- Primitive values
- wrapping
- ...?

Breakdown of Classes, Types, and Functions

Every global definition should be mentioned in one of these categories. It should be described in terms of what it does; its new or extended functionality should be outlined (at a minimum); details and rationale are both plusses.

Fundamental classes (some magic required)

- Object, Function, int, uint, double, decimal, string, boolean, name, namespace
- eval, iterator stuff
- global object

Wrapper classes

- Number, String, Boolean

Performance sensitive classes

- Classes that in most current implementations needs to be implemented in native code to have acceptable performance: Array, RegExp, Math
- parseFloat, toExponential ?
- double/decimal.explode, double/decimal.implode

Language support

- Otherwise trivial stuff that the language depends on (error classes, EnumerableID, ...)

Utility classes

- Stuff that is useful in many programs (Date, Map, Vector, JSON, AnyNumber, AnyString, AnyBoolean, ...); many, many methods (parseInt, ...)
- But not *too much* stuff, because small systems matter, a few fundamental structures that fit the language go a long way, ES4 supports a library ecosystem in various ways, etc.

In general, if a class is predefined then its useful methods are also predefined because the marginal cost of providing more functionality is slight and because there are internal, unexposed APIs that can be used, eg, unicode functionality can be used by eval (for lexing), string methods, regexp methods. (Probably an argument in favor of exposing some unicode stuff, but – then there will be a call to generalize.)