ES6 RegExp Extensibility Design

Allen Wirfs-Brock (allen@wirfs-brock.com)

In response to:

ES6 RegExp Design

RegExp simplification proposal

Daniel Ehrenberg (@littledan), Google

Extension points of ES2015 RegExps

- @@search, @@replace, @@match, @@split
  - Called by String.prototype.search, etc
- get RegExp.prototype.unicode, get RegExp.prototype.multiline, etc
  - Called by RegExp.prototype[@@search], not by RegExp.prototype.exec (uses internal slot)
- RegExp.prototype.exec
  - Called by @@search and friends

Nothing incorrect here but “Extension points” is too vague of a term to use here.

Let’s look at the actual design.

First: Some OO Concepts

- Public Interface/Public contract: The set of methods (and their contract) that are provided for external access to an object's behavior.
  - Objects don't have to be subclasses to be valid implementations of a Public Interface
- Subclass interface/Subclass Contact: The set of methods (and their contract) that are provided to allow a subclass to extend a base class
- Abstract Base Class: A class that provides an extensible abstract implementation of of the Public Interface
  - Abstract Algorithms
  - Kernel Methods
- Concrete Base Class: A class that includes a specific (possibly extensible) internal implementation of the Public Interfacer.

Mapping RegExp to these concepts

RegExp has dual roles as both an abstract base and a concrete base.

Public Interface: exec, test, @@search, @@split, @@match, @@replace, flags, source, global, ignoreCase, etc.

Subclass Interface: exec -- exec is the only kernel method

Abstract algorithm methods: test, flags, @@search, @@split, @@match, @@replace

Concrete methods with built-in pattern matcher dependencies: exec, source, global, ignoreCase, etc

Extension Strategies

- Build a complete new implementation of the public interface.
- Subclass that uses the built-in pattern matcher:
    - For example: logging, memoization, construction time pattern generation/modification
    - Over-ride the exec method (and possibly the constructor)
    - All abstract algorithms and concrete base methods work as inherited
- Subclass that replaces/extends the built-in pattern matcher
    - Over--ride exec method and constructor to provide alternative matcher
    - May need to over-ride other concrete base methods
    - Can still use inherited abstract algorithm methods

Proposal: Reduce the number of extension points

- What remains: @@search, @@replace, @@match, @@split
- Getters still present, but internal algorithms use internal slots
  - Yes, the use of Get to access global and sticky within RegExpBuiltinExe is a bug
- RegExpBuiltinExec is called directly, not RegExp.prototype.exec
  - No, that would destroy the role of the abstract algorithm methods and essentially force every subclass to reimplement: test, flags, @@search, @@split, @@match, @@replace
  - Many of those are complex algorithms that are difficult to implement correctly.
  - It makes simple subclass extensions such as logging or memoization impractical

Proposal: Reduce the number of extension points

- In favor of of the ES2015 spec version
  - RegExp subclasses can replace just `exec` and change all string functions
  - RegExp methods can be retargeted to other classes/objects without internal slots

Yes, those are exactly the design goals for RegExp subclassing. The ES6 RegExp specification achieves these goals using OO extensible design best practices.

ES6 RegExp - Legacy Implementation Reuse

- There are no (intentional) changes to the ES5 specified semantics of built-in RegExp or the String match, search, split, replace methods
- With appropriate guards, existing implementations of String match, search, split, replace methods should be usable in an ES6 implementation
  - Normal sort of guards: is this a built-in RegExp instance with an untampered prototype?
- Any unintentional specification changes that prevent this are bugs that should be fixed.

Proposal: Reduce the number of extension points

- In favor of fewer extension points
  - Reduce implementation complexity
    - Complicated and brittle to put checks for optimizable case
    - This sort of complexity belongs at the implementation level, it should not be a burden placed on everyday ES programmers.
  - A RegExp subclassing framework could be in ES, probably a few hundred lines
    - Framework would implement ES2015 definition of RegExp.prototype[@@search], etc
    - No special support from builtin RegExp class needed
    - Why should builtin RegExp class have this complexity, if it could be done in ECMAScript more easily?

Such a framework is already specified in ES6 based upon OO best practices. It is up to implementations to decide how to implement it and how to optimize. Implementations should feel free to self-host the abstract algorithm methods using ECMAScript and apply their JIT optimizations to them.

Proposal: Reduce the number of extension points

- If we get a concise, usable, spec-compliant RegExp subclassing framework
    - Could we simplify the RegExp spec in future versions?

- Web-compatible to revert--no browser ships fully-compliant new semantics

We don’t need a redesign. We need implementers to report issues they find in the current specification so they can be corrected.