

The Spirit of JavaScript

What's the class of the problem?

When seeking to model a complex language formally, compromises must often be made. Many systems have evolved organically, based on need rather than design, and as such constructing a simple framework of concepts that captures all the nuances of a language is often untenable; complete formalisms reduce to some idealistic core concept, surrounded by myriad “special-cases” (or “hacks”). To make the formalism tractable for both use and analysis, we would prefer to focus on just the “core”, that captures the part of the language we are interested in, hopefully without rendering extension of our work to the full system impossible.

So, what are we interested in? When it comes to a small part of a language, say, the Java garbage collector, it seems fairly easy to pick the subset of the language semantics we care about; memory allocation, reference holding and scope issues. We would not be particularly bothered if we didn't model all the possible looping constructions, or esoteric features like “inner classes”; we know an interested party can add these things later, perhaps with special cases of our model, or by tedious extension of it.

What's our specific problem?

If we set our goals wider, it is much less obvious what we can remove from our mode. The problem before us now is “Web Browser JavaScript”. Not “The DOM library of JavaScript” or “Function calling in JavaScript” - just “Web Browser JavaScript”, a monolithic language and ecosystem which literally evolved from a handful of people adding what they thought was needed to a product that happened to become popular. It is a language of accident, of pragmatism, and a key instance of “Right place, right time” winning out over extensive design. We cannot, in one sitting, capture in one formalism the entirety of JavaScript. Where do we draw the line?

Some aspects seem evidently uninteresting; if we model a for loop, we can syntactically transform any while loop into a semantically identical for loop, and so not be concerned with explicitly supporting “while” semantics. Similarly, minor issues of representation are ripe for omission; operator precedence can be assumed to be disambiguated, strings well formed and so forth. We focus on the meaningful semantic features of the language, the set of features that make JavaScript itself, rather than just a facade over something else. We'll call this the “spirit” of the language.

How do we solve it?

If the solution is to derive a set of syntax and semantics we consider to be the spirit of the language, then we must have a source of features to choose from. Whilst JavaScript is formally defined (ECMA 262-3) many features that are not standardised are de-facto, due to wide implementation and adoption. Moreover, many standardised features are inconsistently implemented, due to their obscure definitions or lack of use. This leads to a situation where a JavaScript expert actually knows a language defined only as the intersection of multiple popular implementation choices. It seems sensible to found our choice of semantics in the abilities of implementations.

Of course, even then, there is not clear choice as to what the true “spirit” is - the language itself admits several programming styles, from procedural imperative, to object oriented imperative, to pure functional. Indeed, the group tasked with the development of the standard recently came out of a year long heated debate, in which proponents of more commonplace features such as a Java like class model, namespaces, early binding and packages were pushing to remodel the soul of the language in a more conventional fashion. Opposing this were those who felt such features were unsuitable for the web environment; for example, early binding being inimical to the dynamic nature of script download. This opposition won through, and JavaScript is undoubtedly a highly dynamic language; very little of a programs environment is not available for alteration.

We’ll first split our problem of “Web Browser JavaScript” into “Web Browser” and “JavaScript”. JavaScript itself has become a general language, not specifically tied to a browser host.

JavaScript

This split allows us to identify the following features of JavaScript. The normal, undefining features are.

- **Simple variables with primitive typing.** Types are String, Boolean, Integer, and the more interesting “Object”
- **Standard imperative assignment**
- **Flow control**
 - Looping constructs, such as “for”, “while”, “do” (with both “break” and “continue” control)
 - Conditional branching, “if” and “switch”
- **Procedures with arbitrary arity and a return value** (all pass-by-value)

These features are unsurprising, being common to virtually every “curly-brace” imperative language. However, the simple presentation masks some key concepts.

- **Functions are first class, and higher order:** Functions are primitive concepts, and can be passed, assigned and created whenever desired.
- **Everything is a property list:** A property list is a set of mappings from strings to arbitrary values. In JavaScript, virtually everything is a property list (for example, a function is a property list with a special callable entry). Property lists can be enumerated using a for/in construct, though some entries can be marked with metadata by the system such that they are either constant, or do not appear during enumeration.
- **Property lists are malleable:** They can be extended or contracted at any point in execution. One can thus create a variable, assign it to a function, and set an arbitrary property on it without issue.
- **Objects are just property list instances, with a prototype chain:** When you create a new object, you effectively call a function which sets up the property list. However, inheritance is achieved by prototype chains - when the runtime cannot find a property, it will ascend the chain to attempt to resolve the property.
- **The set of fixed types is very small:** One cannot introduce new types at runtime - the set of all non-primitive types is merely “Object”, where the property list associated with an instance provides an implicit interface.
- **Virtually everything is mutable.** At runtime, one can redefine concepts that should seemingly be fixed. For example, the system provided “Number” object has a “NaN” property, such that people can compare computational results to a failure state.

However, one can dynamically remove this Number object, and provide a new one with either some new NaN definition, or none at all!

- **Code can be dynamically created and executed:** Using the eval construct, any string can be interpreted as JavaScript and executed in the scope of the eval call
- **Exceptions:** Exceptions are values which are “thrown” to indicate exceptional conditions. At point of throw, normal execution stops and control bubbles up the call stack until the exception is “caught” by a handler. However, during the stack unwinding process, arbitrary code may be executed before each block terminates.

Which of these features should be considered part of the “spirit” of JavaScript?

Web Browser

The key features exposed when hosting JavaScript within a web browser are the interaction with the browsers internal structures, and events. The former, often called the DOM, gives the hosted runtime access to an object that represents the current state of the browsing session; the cookies, current URL, UI information as well as a complex object graph representing the current page. The latter allows JavaScript to respond to a users interaction with the web page, and so provide programs with runtime input from the user. Both are critical to any model of the JS spirit.

Both these items are not merely “in the spirit of”, but essential to any Web Browser embedding of JavaScript. There is still scope for simplification in how deeply we choose to model things. For example, our DOM library may choose to ignore attributes on elements, as they are a simple extension to the normal node structure. The event system may only implement some events, rather than the exhaustive set specified in the standard.