It’s About Time
to Take JavaScript
(More) Seriously

Holger M. Kienle

P

rogramming languages and their features are often hotly debated, mostly on the
grounds of personal beliefs rather than
established facts. In these discussions, Jav-
aScript is left on the sidelines, perhaps
because it’s perceived as a toy with little
real impact. If Perl used to be the Internet’s duct
tape, JavaScript has taken on the role of the In-
ternet’s ugly duckling. Yet, JavaScript deserves a
second look.

Scripting Languages
JavaScript is a scripting language, of course. The
drawbacks and benefits of scripting languages
compared to full-fledged programming languages
have been explored before. In a previous install-
ment of this column, Diomidis Spinellis points
out that with scripting languages, users can bene-
fit from flexible syntax, loose type systems, pow-
eful reflection mechanisms, and shorter build
cycles.  

Scripting languages have proven their use-
fulness in various application areas. Unix relies
heavily on shell scripting to accomplish many
complex tasks. Visual Basic, in combination with
ActiveX, provides a low-entry barrier into the
world of component-based programming. Tcl/Tk
is still popular to rapidly develop GUI-based ap-
lications. And then there’s JavaScript, which has
gained considerable importance in realizing client-
side functionality on Web apps. In a survey on the
most popular programming languages, JavaScript
made it to the Top 10.  

The rise of Web 2.0 has brought with it ever
more sophisticated user interfaces and client-side
browser functionality. As a result, JavaScript has
become a crucial factor for both browser vendors
and Web app developers. Browser vendors are
throwing compiler and interpreter technology at
JavaScript to speed up its execution. This is neces-
sary because sophisticated Web apps designed to
replace desktop apps require a significant amount
of JavaScript code that also must execute at a rea-
sonable speed. Worse, users often have multiple
tabs running Web apps, such as email, calendars,
image viewing and manipulation, and so on.

More than ever, a browser’s JavaScript support
can make or break the user experience. The major
browser vendors are working on new JavaScript
ingines—for example, Google’s V8 and Mozilla’s
TraceMonkey—designed for fast execution using
techniques such as just-in-time (JIT) compilation
and efficient garbage collection. With JavaScript’s
performance, history is about to repeat itself. A
newly introduced language often has to struggle
with performance issues, which prompts critics to
proclaim its doom. Maturing compiler techniques
and Moore’s law then go hand-in-hand in chip-
ing away performance limitations to the point
where it’s no longer an issue. This has happened
relatively recently with Java, but also with tech-
niques such as relational algebra, which took more
than a decade to become practical for relational
database systems.  

Coding JavaScript
While JavaScript is mostly written and hand-
tuned by programmers, it may be increasingly
generated by sophisticated tools. For instance, the
Google Web Toolkit (GWT) lets developers write
client-side code in Java that’s then automatically translated to JavaScript for execution. In a sense, JavaScript acts as the Web browser’s virtual machine. GWT lets Java programmers code in their native environment, effectively developing statically typed client-side code. As an added bonus, GWT’s JavaScript code tries to mitigate browser incompatibilities, which is a major headache for Web apps.

In addition to the client side, a wide range of applications also support JavaScript, such as Adobe products, OpenOffice, Scalable Vector Graphics, Lotus Notes, and Google Apps Script. For example, you can include JavaScript into PDFs rendered with Acrobat. You can also customize Acrobat itself by writing JavaScript extensions. Similar to Visual Basic scripting in Microsoft Office, you can use JavaScript in OpenOffice, Lotus Notes, and Google Docs. Thus, having a working knowledge of JavaScript can come in quite handy for little automation tasks. Such scripts tend to start out with a few lines but can quickly grow to a few pages. If you use JavaScript’s features to good effect, the code will be better structured and easier to read and maintain.

Though not readily apparent, JavaScript offers a powerful mix of interesting language features based on functional programming, prototyping, and mutable objects—that is, objects can change their structure at runtime. In JavaScript, a hashtable is expressed with curly braces: `my_rect = { w: 42, h: 43 }`. This hashtable maps two properties, `w` and `h`, to integer values. Actually, since hashtables and objects are the same thing in JavaScript, this can be also read as an object `my_rect`, having two slots containing instance variables. If we want to introduce a method `area` (or replace the `area` later with a different implementation) we can do this with `my_rect.area = function () { return this.w * this.h }`. The function operator produces a (anonymous) function object that’s assigned to the object’s area slot. (If a function is called via an object, then this is bound to the calling object in the function’s body.) JavaScript supports closures, and functions are in fact objects, enabling rudimentary functional programming. In fact, the Functional library provides a convenient API for functional programming that you can try out directly at http://osteele.com/sources/javascript/functional.

What inheritance is to class-based languages, delegation is to prototype-based ones. If a slot is accessed on an object, say `x`, and this slot isn’t found in the object itself, the lookup for `x` is repeated at another object, the object’s prototype. (Mozilla’s JavaScript engine exposes the prototype reference as `__proto__` slot.) The lookup follows the whole chain of prototype objects until it locates `x` or reaches the end of the chain. Coming back to our example, a colored rectangle that delegates to the `my_rect` object can be realized with `my_c_rect = { color: “red”; __proto__: my_rect }`. Now, if `my_c_rect.w` is called, the result will be 42 because the lookup locates `my_rect`’s `x` slot in following the prototype chain. This toy example only conveys the basic idea. Similar to class-based programming, prototype programming derives its power from knowing the right coding idioms and design patterns.

If you want to see a larger example of well-written JavaScript, Douglas Crockford has contributed a chapter in Beautiful Code that walks you through the implementation of a parser that uses Pratt’s top-down operator precedence technique. He summarizes his opinion of JavaScript pointedly when he says that “the language is a mess” but also that it “has some brilliant stuff in it.”

Crockford’s Web site (http://javascript.crockford.com) provides good pointers on JavaScript. For example, he explains how you can mimic class-based inheritance with JavaScript’s prototype mechanism. Another fun site to poke around is http://ejohn.org.

**Lively Kernel**

A number of projects leverage JavaScript to good effect. For example, a group at Sun Labs including Dan Ingalls has developed the Lively Kernel (http://research.sun.com/projects/lively/), which is an interactive platform and Web development environment that runs in a browser window and is implemented entirely in JavaScript. Rich graphics is supported with the Morphics framework (inspired by Squeak) that’s coded in 10,000 lines of JavaScript. Morphics relies on browsers’ native Scalable Vector Graphics (SVG) support for rendering. Leveraging JavaScript’s introspective features, the Lively Kernel comes with a code browser and object inspector that makes it possible to modify the platform and its applications during runtime. For the Lively Kernel, the developers used JavaScript as a general-purpose and systems programming language.

On the downside, the Lively Kernel developers found that JavaScript lacks support for modularity and has no information hiding. Parsers are often too permissive, resulting in bugs that are hard to track down—confusing `0`, `[]`, and `[]` isn’t a good idea. As a band-aid, there’s a static verification tool, jslint. Future language versions should address some of these issues. The latest standard, ECMAScript 5, released December 2009, addresses access control and lets programmers activate strict code rules. JavaScript lacks a rich set of standardized libraries. However, the JavaScript community has developed third-party libraries such as jQuery, the Prototype framework, and script.aculous.us. For a more complete list, see http://en.wikipedia.org/wiki/List_of_JavaScript_libraries. These libraries also address missing functionality in the language proper, such as class-based inheritance and code include features.

On the positive side, when developers coded Lively Kernel, JavaScript turned out to “be a surprisingly expressive language” enabling dense coding. Perhaps also surprising, JavaScript “can be used for developing real applications.” Interestingly, rewriting existing C and Java code into JavaScript is more an act of transliteration than translation.
However, developing applications with JavaScript requires a programming style that’s highly iterative and exploratory where one “proceed[s] step by step, by writing (and immediately testing) each new piece of code.” JavaScript’s name is deceptive in the sense that the development experience of a Java programmer is probably far less applicable than that of a Smalltalk or Lisp coder. Luckily there’s support for this development style in the form of testing frameworks (QUnit and JSUnit) and debuggers (Firebug).

While JavaScript may be an ugly duckling, it has a few spots where it shines. Lively Kernel’s developers believe that with more advanced JavaScript applications and more sophisticated libraries, negative perceptions of JavaScript will improve. Regardless of whether you like it, expect to see more JavaScript—or at least to execute more of it in your browser—as we move down the Web 2.0 road. As Ingalls said, “just for fun, let’s take JavaScript seriously,” and let’s see where it leads us.

References

Holger M. Kienle is a postdoctoral researcher in software evolution at Mälardalen University and the University of Victoria. Contact him at h.kienle@acm.org.