JavaScript 2.0:
Evolving a Language for Evolving Systems

Waldemar Horwat
waldemar@acm.org

Abstract

JavaScript 2.0 is the next major revision of the JavaScript language. Also known as ECMAScript Edition 4, it is being standardized by the ECMA organization. This paper summarizes the needs that drove the revision in the language and then describes some of the major new features of the language to meet those needs — support for API evolution, classes, packages, object protection, dynamic types, and scoping. JavaScript is a very widely used language, and evolving it presented many unique challenges as well as some opportunities. The emphasis is on the rationale, insights, and constraints that led to the features rather than trying to describe the complete language.

1 Introduction

1.1 Background

JavaScript [6][8] is a web scripting language invented by Brendan Eich at Netscape. This language first appeared in 1996 as JavaScript 1.0 in Navigator 2.0. Since then the language has undergone numerous additions and revisions [6], and the most recent released version is JavaScript 1.5.

JavaScript has been enormously successful — it is more than an order of magnitude more widely used than all other web client languages combined. More than 25% of web pages use JavaScript.

JavaScript programs are distributed in source form, often embedded inside web page elements, thus making it easy to author them without any tools other than a text editor. This also makes it easier to learn the language by examining existing web pages.

There is a plethora of synonymous names for JavaScript. JavaScript, JScript, and ECMAScript are all the same language. JavaScript was originally called LiveScript but was renamed to JavaScript just before it was released. JavaScript is not related to Java, although the two language implementations can communicate with each other in Netscape browsers through an interface called LiveConnect.

JavaScript as a language has computational facilities only — there are no input/output primitives defined within the language. Instead, each embedding of JavaScript within a particular environment provides the means to interact with that environment. Within a web browser JavaScript is used in conjunction with a set of common interfaces, including the Document Object Model [11], which allow JavaScript programs to interact with web pages and the user. These interfaces are described by separate standards and are not part of the JavaScript language itself. This paper concentrates on the JavaScript language rather than the interfaces.

1.2 Standardization

After Netscape released JavaScript in Navigator 2.0, Microsoft implemented the language, calling it JScript, in Internet Explorer 3.0. Netscape, Microsoft, and a number of other companies got together and formed the TC39 committee in the ECMA standards organization [2] in order to agree on a common definition of the language. The first ECMA standard [3], calling the language ECMAScript, was adopted by the ECMA general assembly in June 1997 as the ECMA-262 standard. The second edition of

this standard, ECMA-262 Edition 2 [4], consisted mainly of editorial fixes gathered in the process of making the ECMAScript ISO standard 16262. The third edition of the ECMAScript standard [5] was adopted in December 1999 and added numerous new features, including regular expressions, nested functions and closures, array and object literals, the `switch` and `do-while` statements, and exceptions. JavaScript 1.5 fully implements ECMAScript Edition 3.

I've been involved at Netscape with both the implementation and standardization of JavaScript since 1998. I wrote parts of the ECMAScript Edition 3 standard and am currently the editor of the draft ECMAScript Edition 4 standard.

In Editions 1 and 2, the ECMA committee standardized existing practice, as the language had already been implemented by Netscape, and Microsoft closely mirrored that implementation. In Edition 3, the role of the committee shifted to become more active in the definition of new language features before they were implemented by the vendors; without this approach, the vendors' implementations would have quickly diverged. This role continues with Edition 4, and, as a result, the interesting language design discussions take place mainly within the ECMA TC39 (now TC39TG1) working group.

This paper presents the results of a few of these discussions. Although many of the issues have been settled, Edition 4 has not yet been approved or even specified in every detail. It is still likely to change and should definitely be considered a preliminary draft.

1.3 Outline

Section 2 gives a brief description of the existing language JavaScript 1.5. Section 3 summarizes the motivation behind JavaScript 2.0. Individual areas and decisions are covered in subsequent sections: types (Section 4); scoping and syntax issues (Section 5); classes (Section 6); namespaces, versioning, and packages (Section 7); and attributes and conditional compilation (Section 8). Section 9 concludes.

2 JavaScript 1.5

JavaScript 1.5 (ECMAScript Edition 3) is an object-based scripting language with a syntax similar to C and Java. Statements such as `if`, `while`, `for`, `switch`, and `throw/try/catch` will be familiar to C/C++ or Java programmers. Functions, declared using the `function` keyword, can nest and form true closures. For example, given the definitions

```
function square(x) {
  return x*x;
}

function add(a) {
  return function(b) {
    return a+b;
  }
}
```

evaluating the expressions below produces the values listed after the □ symbols:

```
square(5) □ 25
var f = add(3);
var g = add(6);
f(1) □ 4;
g(5) □ 11;
```

A function without a `return` statement returns the value `undefined`.

Like Lisp, JavaScript provides an `eval` function that takes a string and compiles and evaluates it as a JavaScript program; this allows self-constructing and self-modifying code. For example:

```
eval("square(8)+3") □ 67
eval("square = f") □ The source code for function f
square(2) □ 5
```

2.1 Values and Variables

The basic values of JavaScript 1.5 are numbers (double-precision IEEE floating-point values including +0.0, –0.0, +∞, –∞, and NaN), booleans (`true` and `false`), the

special values `null` and `undefined`, immutable Unicode strings, and general objects, which include arrays, regular expressions, dates, functions, and user-defined objects. All values have unlimited lifetime and are deleted only via garbage collection, which is transparent to the programmer.

Variables are not statically typed and can hold any value. Variables are introduced using `var` declarations as in:

```
var x;
var y = z+5;
```

An uninitialized variable gets the value `undefined`. Variable declarations are lexically scoped, but only at function boundaries — all declarations directly within a function apply to the entire function, even above the point of declaration. Local blocks do not form scopes. If a function accesses an undeclared variable, it is assumed to be a global variable. For example, in the definitions

```
function init(a) {
  b = a;
}

function strange(s, t) {
  a = s;
  if (t) {
    var a;
    a = a+a;
  }
  return a+b;
}
```

function `strange` defines a local variable `a`. It doesn't matter that the `var` statement is nested within the `if` statement — the `var` statement creates `a` at the beginning of the function regardless of the value of `t`.

At this point evaluating

```
strange("Apple ", false)
```

signals an error because the global variable `b` is not defined. However, the following statements evaluate successfully because `init` creates the global variable `b`:

```
init("Hello") ⇒ undefined
strange("Apple ", false) ⇒
"Apple Hello"
strange(20, true) ⇒
"40Hello"
```

The last example also shows that + is polymorphic — it adds numbers, concatenates strings, and, when given a string and a number, converts the number to a string and concatenates it with the other string.

2.2 Objects

JavaScript 1.5 does not have classes; instead, general objects use a prototype mechanism to mimic inheritance. Every object is a collection of name-value pairs called properties, as well as a few special, hidden properties. One of the hidden properties is a prototype link[1] which points to another object or `null`.

When reading property *p* of object *x* using the expression *x*`.`*p*, the object *x* is searched first for a property named *p*. If there is one, its value is returned; if not, *x*'s prototype (let's call it *y*) is searched for a property named *p*. If there isn't one, *y*'s prototype is searched next and so on. If no property at all is found, the result is the value `undefined`.

When writing property *p* of object *x* using the expression *x*`.`*p* = *v*, a property named *p* is created in *x* if it's not there already and then assigned the value *v*. *x*'s prototype is not affected by the assignment. The new property *p* in *x* will then shadow any property with the same name in *x*'s prototype and can only be removed using the expression `delete` *x*`.`*p*.

A property can be read or written using an indirect name with the syntax *x*`[`*s*`]`, where *s* is an expression that evaluates to a string (or a value that can be converted into a string) representing a property name. If *s* contains the string `"blue"`, then the expression *x*`[`*s*`]` is equivalent to *x*`.blue`. An array is

[1] For historical reasons in Netscape's JavaScript this hidden prototype link is accessible as the property named `__proto__`, but this is not part of the ECMA standard.

an object with properties named "0", "1", "2", …, "576", etc.; not all of these need be present, so arrays are naturally sparse.

An object is created by using the new operator on any function call: new *f*(*args*). An object with no properties is created before entering the function and is accessible from inside the function via the this variable.

The function *f* itself is an object with several properties. In particular, *f*.prototype points to the prototype that will be used for objects created via new *f*(*args*).[2] An example illustrates these concepts:

```
function Point(px, py) {
  this.x = px;
  this.y = py;
}

a = new Point(3,4);
origin = new Point(0,0);

a.x ⇒ 3
a["y"] ⇒ 4
```

The prototype can be altered dynamically:

```
Point.prototype.color =
"red";

a.color ⇒ "red"
origin.color ⇒ "red"
```

The object a can shadow its prototype as well as acquire extra properties:

```
a.color = "blue";
a.weight = "heavy";

a.color ⇒ "blue"
a.weight ⇒ "heavy"
origin.color ⇒ "red"
origin.weight ⇒ undefined
```

Methods can be attached to objects or their prototypes. A method is any function. The method can refer to the object on which it was invoked using the this variable:

```
function Radius() {
  return Math.sqrt(
    this.x*this.x +
    this.y*this.y);
}
```

The following statement attaches Radius as a property named radius visible from any Point object via its prototype:

```
Point.prototype.radius =
Radius;

a.radius() ⇒ 5
```

The situation becomes much more complicated when trying to define a prototype-based hierarchy more than one level deep. There are many subtle issues [9], and it is easy to define one with either too much or too little sharing.

2.3 Permissiveness

JavaScript 1.5 is very permissive — strings, numbers, and other values are freely coerced into one another; functions can be called with the wrong number of arguments; global variable declarations can be omitted; and semicolons separating statements on different lines may be omitted in unambiguous situations. This permissiveness is a mixed blessing — in some situations it makes it easier to write programs, but in others it makes it easier to suffer from hidden and confusing errors.

For example, nothing in JavaScript distinguishes among regular functions (square in the examples above), functions intended as constructors (Point), and functions intended as methods (Radius). JavaScript lets one call Point defined above as a function (without new and without attaching it to an object),

```
p = Point(3)
```

which creates *global* variables x and y if they didn't already exist (or overwrites them if they did) and then writes 3 to x and

[2] Using the notation from the previous footnote, after *o* = new *f*(*args*), *o*.__proto__ == *f*.prototype. *f*.prototype is not to be confused with function *f*'s own prototype *f*.__proto__, which points to the global prototype of functions Function.prototype.

`undefined` to `y`. The variable `p` gets the value `undefined`. Obvious, right? (If this is obvious, then you've been spending far too much time reading language standards.)[3]

2.4 Exploring Further

This is only a brief overview of JavaScript 1.5. See a good reference [6] for the details. To get an interactive JavaScript shell, type `javascript:` as the URL in a Netscape browser or download and compile the source code for a simple stand-alone JavaScript shell from [8].

3 JavaScript 2.0 Motivation

JavaScript 2.0 is Netscape's implementation of the ECMAScript Edition 4 standard currently under development. The proposed standard is motivated by the need to achieve better support for *programming in the large* as well as fix some of the existing problems in JavaScript (section 5).

3.1 Programming in the Large

As used here, programming in the large does not mean writing large programs. Rather, it refers to:

- Programs written by more than one person
- Programs assembled from components (packages)
- Programs that live in heterogeneous environments
- Programs that use or expose evolving interfaces
- Long-lived programs that evolve over time

Many applications on the web fall into one or more of these categories.

[3] The reason that global variables `x` and `y` got created is that when one doesn't specify a `this` value when calling a function such as `Point`, then `this` refers to the global scope object; thus `this.x = px` creates the global variable `x`.

3.2 Mechanisms

A package facility (separable libraries that export top-level definitions — see section 7) helps with some of the above requirements but, by itself, is not sufficient. Unlike existing JavaScript programs which tend to be monolithic, packages and their clients are typically written by different people at different times. This presents the problem of the author or maintainer of a package not having access to all of its clients to test the package, or, conversely, the author of a client not having access to all versions of the package to test against — even if the author of a client could test his client against all existing versions of a package, he is not able to test against future versions. Merely adding packages to a language without solving these problems would not achieve robustness; instead, additional facilities for defining stronger boundaries between packages and clients are needed.

One approach that helps is to make the language more disciplined by adding optional types and type-checking (section 4). Another is a coherent and disciplined syntax for defining classes (section 6) together with a robust means for versioning of classes. Unlike JavaScript 1.5, the author of a class can guarantee invariants concerning its instances and can control access to its instances, making the package author's job tractable. Versioning (section 7) and enforceable invariants simplify the package author's job of evolving an already-published package, perhaps expanding its exposed interface, without breaking existing clients. Conditional compilation (section 8) allows the author of a client to craft a program that works in a variety of environments, taking advantage of optional packages if they are provided and using workarounds if not.

To work in multi-language environments, JavaScript 2.0 provides better mappings for data types and interfaces commonly exposed by other languages. It includes support for classes as well as previously missing basic types such as `long`.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.