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`.

3.3 Non-Goals

JavaScript 2.0 is intended for a specific niche of scripting languages. It is meant to be a glue language. It is not meant to be:

- a high-performance language
- a language for writing general-purpose applications such as spreadsheets, word processors, etc.
- a language for writing huge programs
- a stripped-down version of an existing language

Although many of the facilities provided improve performance, that by itself is not their reason for inclusion in the language.

4 Type System

JavaScript 2.0 supports the notion of a type, which can be thought of as a subset of all possible values. There are some built-in types such as `Object`, `Number`, and `String`; each user-defined class (section 6) is also a type.

The root of the type hierarchy is `Object`. Every value is a member of the type `Object`. Unlike in JavaScript 1.5, there is no real distinction between primitive values and objects[4].

Unlike in C and Java, types are first-class values. Type expressions are merely value expressions that evaluate to values that are types; therefore, type expressions use the same syntax as value expressions.

4.1 Type Declarations

Variables in JavaScript 2.0 can be typed using the syntax

```
var v:type = value;
```

where *v* is the name of the variable and *type* is a constant expression that evaluates to a type. Types can also be attached to function parameters and results.

A variable declared with a type is guaranteed to always hold an element of that type[5]. Assigning a value to that variable coerces the value to the type or generates an error if the coercion is not allowed. To catch errors, such coercions are less permissive than JavaScript 1.5's coercions.

4.2 Strong Dynamic Typing

JavaScript 2.0 is *strongly typed* — type declarations are enforced. On the other hand, JavaScript 2.0 is not *statically typed* — the compiler does not verify that type errors cannot occur at run time. To illustrate the difference, consider the class definitions below, which define a class `A` with instance variable `x` and a subclass `B` of `A` with an additional instance variable `y`:

```
class A {
  var x;
}

class B extends A {
  var y;
}
```

Given the above, the following statements all work as expected:

```
var a:A = new A;
var b:B = new B;
a = b;
var o = new A;
```

An untyped variable such as `o` is considered to have type `Object`, so it admits every value. The following statements, which would be errors in a statically typed language, also execute properly because the run-time values being assigned are of the proper type:

[4] The bizarre JavaScript 1.5 dichotomy between `String`, `Number`, and `Boolean` values and `String`, `Number`, and `Boolean` objects is eliminated, although an implementation may preserve it as an optional language extension for compatibility. All JavaScript 2.0 values behave as though they are objects — they have methods and properties — although some of the more important classes such as `String`, `Number`, etc. are `final` and don't allow the creation of dynamic properties, so their instances can be transparently implemented as primitives.

[5] Actually, the rule is that successfully reading a variable always returns an element of the variable's type. This is because a variable may be in an uninitialized state, in which case trying to read it generates an error.

```
b = a;
a = o;
```

On the other hand, assigning `b = o` generates a run-time error because `o` does not currently contain an instance of `B`.

Because JavaScript is not statically typed, function `sum` below also compiles correctly; it would be a compile-time error in a statically typed language because the compiler could not statically prove that `c` will have a property named `y`.[6]

```
function sum(c:A) {
  return c.x + c.y;
}
```

The assignment to `z1` will execute successfully, while the assignment to `z2` will generates a run-time error when trying to look up `c.y`:[7]

```
var z1 = sum(new B);
var z2 = sum(new A);
```

The declaration `c:A` inside `sum` is still enforced — it requires that the argument passed to `sum` must be a member of type `A`; thus, an attempt to call `sum` on an instance of some class `C` unrelated to `A` would generate an error even if that instance happened to have properties `x` and `y`.

The general principle here is that only the actual run-time type of an expression's value matters — unlike statically typed languages such as C++ and Java, JavaScript 2.0 has no concept of the static type of an expression.

[6] If class `A` were final, a smart JavaScript compiler could issue a compile-time error for function `sum` because it could prove that no possible value of `c` could have a property named `y`. The difference here is that a compiler for a statically typed language will issue an error if it cannot prove that the program will work without type errors. A compiler for a dynamically typed language will issue an error only if it can prove that the program cannot work without type errors; strong typing is ensured at run time.

[7] Unlike with prototype-based objects, by default an attempt to refer to a nonexistent property of a class instance signals an error instead of returning `undefined` or creating the property. See section 6.

4.3 Rationale

Why doesn't JavaScript 2.0 support static typing? Although this would help catch programmer errors, it would also dramatically change the flavor of the language. Many of the familiar idioms would no longer work, and the language would need to acquire the concept of interfaces which would then have to be used almost everywhere. Followed to the logical conclusion, the language would become nearly indistinguishable from Java or C#; there is no need for another such language.

Another common question is why JavaScript 2.0 uses the colon notation for type annotation instead of copying the C-like syntax. Embarrassingly, this is a decision based purely on a historical standards committee vote — this seemed like a good idea at one time. There is no technical reason for using this syntax, but it's too late to reverse it now (implementations using this syntax have already shipped), even though most of the people involved with it admit the syntax is a mistake.

5 Scoping and Strict Mode

JavaScript 1.5 suffers from a number of design mistakes (see sections 2.1 and 2.3 for some examples) that are causing problems in JavaScript 2.0. One of the problems is that all `var` declarations inside a function are *hoisted*, which means that they take effect at the very beginning of the function even if the `var` declarations are nested inside blocks. Furthermore, duplicate `var` declarations are merged into one. This is fine for untyped variables, but what should happen for typed variables? What should the interpretation of the following function be?

```
function f(a) {
  if (a) {
    var b:String = g();
  } else {
    var b:Number = 17;
  }
}
```

Using JavaScript 1.5 rules would interpret the function as the following, which would be an error because now `b` has two different types:

```
function f(a) {
  var b:String;
  var b:Number;
  if (a) {
    b = g();
  } else {
    b = 17;
  }
}
```

JavaScript 2.0 also introduces the notion of `const`, which declares a constant rather than a variable. If `b` were a `const` instead of a `var`, then even if the two declarations had the same type then it would be undesirable to hoist it:

```
function f(a) {
  if (a) {
    const b = 5;
  } else {
    const b = 17;
  }
}
```

should not become:

```
function f(a) {
  const b;
  if (a) {
    b = 5;
  } else {
    b = 17;
  }
}
```

For one thing, the latter allows `b` to be referenced after the end of the `if` statement.

To solve these problems while remaining compatible with JavaScript 1.5, JavaScript 2.0 adopts block scoping with one exception: `var` declarations without a type and in *non-strict mode* (see below) are still hoisted to the top of a function. `var` declarations with a type are not hoisted, `const` declarations are not hoisted, and declarations in strict mode are not hoisted. To help catch errors, a block nested inside another block within a function may not redeclare a local variable. Moreover, if a block declares a local variable named *x*, then an outer block in the same function may not refer to a global variable named *x*. Thus, the following code is in error because the `return` statement is not permitted to refer to the global `x`:

```
var x = 3;

function f(a) {
  if (a) {
    var x:Number = 5;
  }
  return x;
}
```

5.1 Strict Mode

Some of JavaScript 1.5's quirks can't be corrected without breaking compatibility. For these JavaScript 2.0 introduces the notion of a *strict mode* which turns off some of the more troublesome behavior. In addition to making all declarations lexically scoped, strict mode does the following:

- **Variables must be declared** — misspelled variables no longer automatically create new global variables.
- **Function declarations are immutable** (JavaScript 1.5 treats any function declaration as declaring a variable that may be replaced by another function or any other value at any time).
- Function calls are checked to make sure that they provide the **proper number of arguments**. JavaScript 2.0 provides an explicit way of declaring functions that take optional, named, or variable amounts of arguments.
- **Semicolon insertion changes** — line breaks are no longer significant in strict-mode JavaScript 2.0 source code. Line breaks no longer turn into semicolons (as they do in some places in JavaScript 1.5), and they are now allowed anywhere between two tokens.

Strict and non-strict parts may be mixed freely within a program. For compatibility, the default is non-strict mode.

6 Classes

In addition to the prototype-based objects of JavaScript 1.5, JavaScript 2.0 supports class-based objects. Class declarations are best illustrated by an example:

```
class Point {
  var x:Number;
  var y:Number;

  function radius() {
    return Math.sqrt(
      x*x + y*y);
  }

  static var count = 0;
}
```

A class definition is like a block in that it can contain arbitrary statements that are evaluated at the time execution reaches the class; however, definitions inside the class define instance (or class if preceded with the `static` attribute) members of the class instead of local variables. Classes can inherit from other classes, but multiple inheritance is not supported.

Classes can co-exist with prototype-based objects. The syntax to read or write a property (*object*`.`*property*) is the same regardless of whether the object is prototype or class-based. By default, accessing a nonexistent property of a class instance is an error, but if one places the attribute `dynamic` in front of the class declaration then one can create new dynamic properties on that class's instances just like for prototype-based objects.

6.1 Rationale

There are a number of reasons classes were added to JavaScript 2.0:

- Classes provide **stronger and more flexible abstractions** than prototypes. A class can determine the pattern of members that each instance must have, control the creation of instances, and control both its usage and overriding interfaces. Furthermore, a JavaScript 2.0 class can enforce these rules without cooperation from its clients, which allows well-constructed classes to rely on their invariants regardless of what their clients do.
- Classes provide a good basis for **versioning and access control** (section 7).
- **Prototype-based languages naturally evolve classes anyway** by convention, typically by introducing dual hierarchies that include prototype and traits objects [1]. Placing classes in the language makes the convention uniform and enforceable.[8]
- **Complexity of prototypes**. Few scripters are sophisticated enough to correctly create a multi-level prototype-based hierarchy in JavaScript 1.5. In fact, this is difficult even for moderately experienced programmers.
- The class syntax is much more **self-documenting** than analogous JavaScript 1.5 prototype hierarchies.
- Classes as a primitive in the language provide a valuable means of **reflecting other languages' data structures** in JavaScript 2.0 and *vice versa*.
- Classes are one of the **most-requested** features in JavaScript.

Introducing two means of doing something (classes and prototypes) always carries some burden of having to choose ahead of time which means to use for a particular problem and the subsequent danger of needing to recover from having made the wrong choice. However, it's likely that at some point in the future most programmers will use classes exclusively and not even bother to learn prototypes. To make recovery easier, the syntax for routine usage of classes and prototypes is identical, so changing one to the other only requires changing the declaration.

[8] An earlier JavaScript 2.0 proposal actually reflected a class's members via prototypes and traits objects and allowed any class instance to serve as a base for prototype inheritance and *vice versa*. That proposal was dropped because it made language implementation much more complex than desired and required the authors of classes to think about not only constructors but also cloners just in the rare case that a client used the classes' instances as prototypes.

To keep the language simple, there is no notion of Java-like interfaces. Unlike in Java, these are not necessary for polymorphism because JavaScript 2.0 is dynamically typed.

7 Namespaces, Versioning, and Packages

7.1 Packages

A JavaScript 2.0 package is a collection of top-level definitions declared inside a `package` statement. An `import` statement refers to an existing package and makes the top-level definitions from that package available. The exact scheme used to name and locate existing packages is necessarily dependent on the environment in which JavaScript 2.0 is embedded and will be defined and standardized independently as needed for each kind of embedding (browsers, servers, standalone implementations, etc.).

7.2 Versioning Issues

As a package evolves over time it often becomes necessary to change its exported interface. Most of these changes involve adding definitions (top-level or class members), although occasionally a definition may be deleted or renamed. In a monolithic environment where all JavaScript source code comes preassembled from the same source, this is not a problem. On the other hand, if packages are dynamically linked from several sources then versioning problems are likely to arise.

One of the most common avoidable problems is collision of definitions. Unless this problem is solved, an author of a package will not be able to add even one definition in a future version of his package because that definition's name could already be in use by some client or some other package that a client also links with. This problem occurs both in the global scope and in the scopes of classes from which clients are allowed to inherit.

7.3 Scenario

Here's an example of how such a collision can arise. Suppose that a package provider creates a package called `BitTracker` that exports a class `Data`. This package becomes so successful that it is bundled with all web browsers produced by the BrowsersRUs company:

```
package BitTracker {

class Data {
  var author;
  var contents;
  function save() {…}
}

function store(d) {
  …
  storeOnFastDisk(d);
}
}
```

Now someone else writes a client web page *W* that takes advantage of `BitTracker`. The class `Picture` derives from `Data` and adds, among other things, a method called `size` that returns the dimensions of the picture:

```
import BitTracker;

class Picture extends
      Data {
  function size() {…}
  var palette;
};

function orientation(d) {
  if (d.size().h >=
      d.size().v)
    return "Landscape";
  else
    return "Portrait";
}
```

The author of the `BitTracker` package, who hasn't seen *W*, decides in response to customer requests to add a method called `size` that returns the number of bytes of data in a `Data` object. He then releases the new and improved `BitTracker` package.

BrowsersRUs includes this package with its latest Navigator 17.0 browser:

```
package BitTracker {

class Data {
  var author;
  var contents;
  function size() {…}
  function save() {…}
}

function store(d) {
  …
  if (d.size() > limit)
    storeOnSlowDisk(d);
  else
    storeOnFastDisk(d);
}
}
```

An unsuspecting user *U* upgrades his old BrowsersRUs browser to the latest Navigator 17.0 browser and a week later is dismayed to find that page *W* doesn't work anymore. *U*'s grandson tries to explain to *U* that he's experiencing a name conflict on the `size` methods, but *U* has no idea what the kid is talking about. *U* attempts to contact the author of *W*, but she has moved on to other pursuits and is on a self-discovery mission to sub-Saharan Africa. Now *U* is steaming at BrowsersRUs, which in turn is pointing its collective finger at the author of `BitTracker`.

Note that this name collision occurs inside a class and is much more insidious than merely a conflict among global declarations from imported packages.

7.4 Solutions

How could the author of `BitTracker` have avoided this problem? Simply choosing a name other than `size` wouldn't work, because there could be some other page *W2* that conflicts with the new name. There are several possible approaches:

- **Naming conventions**. Each defined name could be prefixed by the full name of the party from which this definition originates. Unfortunately, this would get tedious and unnecessarily impact casual uses of the language. Furthermore, this approach is impractical for names of methods because it is often desirable to share the same method name across several classes to attain polymorphism; this would not be possible if Netscape's objects used `com_netscape_length` while MIT's objects all used `edu_mit_length`.
- **Explicit imports**. Each client package could be required to import every external name it references. This works reasonably well for global names but becomes tedious for the names of class members, which would have to be imported separately for each class.
- **Resolve names at compile time**. Java and C# resolve the references of names to the equivalents of vtable slots at compile time. This way `size` inside `store` can resolve to something other than `size` inside `orientation`. Unfortunately, this approach works only for statically typed languages. Moreover, this approach relies on object code rather than source code being distributed — the ambiguity is still present in the source code, and it is only the extra data inserted by past compilation of the client against an older version of the package that resolves it.
- **Versions**. Package authors could mark the names they export with explicit versions. A package's developer could introduce a new version of the package with additional names as long as those names were made invisible to clients expecting to link with prior versions.

JavaScript 2.0 follows the last approach. It is the most desirable because it places the smallest burden on casual users of the language, who merely have to import the packages they use and supply the current version numbers in the import statements. A package author has to be careful not to disturb the set of visible prior-version definitions when releasing an updated package, but

authors of dynamically linkable packages tend to be much more sophisticated than casual users of the language.

7.5 Namespaces

JavaScript 2.0 employs *namespaces* to provide safe versioning. A package can define and export several namespaces, each of which provides a different view of the package's contents. Each namespace corresponds to a version of the package's API.

A JavaScript 2.0 namespace is a first-class value that is merely a unique token and has no members, internal structure, or inheritance. JavaScript namespaces are not related to C++ namespaces. On the other hand, the designers of other dynamic languages such as Smalltalk have independently run into the same versioning problem and come up with a solution similar to JavaScript 2.0's (for example, "Selector Namespaces" in [10]).

Each JavaScript 2.0 name is actually an ordered pair *namespace*`::`*identifier*, where *namespace* is a simple expression that evaluates to a namespace value. When a name is defined without a namespace, the namespace defaults to the predefined namespace `public`.

A `use namespace(`*n*`)` statement allows unqualified access within a scope to identifiers qualified with namespace *n*. There is an implicit `use namespace(public)` statement around the whole program. For convenience, a namespace may also be specified when importing a package.

Given namespaces, the author of `BitTracker` can release the updated package while hiding the `size` method from existing clients that don't expect to see it:

```
package BitTracker {

explicit namespace v2;
use namespace(v2);

class Data {
  var author;
  var contents;
  v2 function size() {…}
  function save() {…}
}

function store(d) {
  …
  if (d.size() > limit)
    storeOnSlowDisk(d);
  else
    storeOnFastDisk(d);
}
}
```

If the client *W* isn't updated, then it will not be aware that `BitTracker`'s `size` exists and will be able to refer to its own `size` method. If the author of *W* later wants to revise her web page to also refer to `BitTracker`'s `size`, then she can either explicitly refer to `v2::size` or rename her `size` to some other name and then import `v2`.[9]

7.6 Private and Internal

In addition to providing access control among packages, namespaces also work well within a package. In fact, early in the development of JavaScript 2.0 it became apparent that the notions of `private` and `internal` (visible inside a package only) are simply special cases of namespaces. Each class has a predefined `private` namespace that can be referenced using the `private` keyword inside that class and that

[9] This description is glossing over how the author of `BitTracker` keeps the name of the namespace itself `v2` from colliding with some other global definition inside the client *W*. The basic explanation is that the `explicit` attribute keeps `v2` itself from being imported by default. If a client knows that `v2` is there and won't cause a conflict, then it can explicitly request `v2` to be imported; there is a convenient syntax for doing that using an `import` statement. See [7] for the details.

is `use`'d only inside that class. Different classes have independent `private` namespaces.

Namespaces offer a finer granularity of permissions — a class can have some `private` members visible only to itself, but it can also define members defined in a custom namespace visible to some but not all of its users.

7.7 Property Lookup

Namespaces control the behavior of looking up either a qualified property *n*`::`*p* or an unqualified property *p* of an object *o*. It may appear that there are many ways of defining this lookup process, but in fact only one way allows for reliable versioning. The resulting lookup process appears counterintuitive at first but is in fact the correct one and deserves a closer look.

The process is illustrated by a few examples. Consider first the simple case of a hierarchy of three classes:

```
class A {
  public function f()
    {return "fA"}
}

class B extends A {
  public function f()
    {return "fB"}
}

class C extends B {
  public function f()
    {return "fC"}
}

c = new C;
```

If one calls `c.f()`, one would expect to, and does in fact, get `"fC"`, the most derived definition of `f`. This is the essence of object-oriented semantics.

Now let's alter the example by putting `A` and `B`'s definitions of `f` into a namespace `N`:

```
class A {
  N function f()
    {return "fA"}
}

class B extends A {
  N function f()
    {return "fB"}
}

class C extends B {
  public function f()
    {return "fC"}
}

c = new C;
```

Now, if one calls `c.f()`, the result ought to depend on whether a `use namespace(N)` is lexically in effect around the expression `c.f()`. If it is not, then the behavior is simple — only the `f` defined by `C` is visible, so once again the result is `"fC"`.

If `use namespace(N)` is lexically in effect around the expression `c.f()`, then all three definitions of `f` are visible. It would be tempting to choose the most derived one (`"fC"`), but this would be incorrect because:

- This is analogous to the `BitTracker` scenario. Class `C` might have defined function `f` first and classes `A` and `B` later evolved to define their own, hidden `f`. To make this work, any code in the lexical scope of a `use namespace(N)` must not have its meaning hijacked by anything class `C` does.
- Suppose that `N` is `private` or `internal` instead. Class `C` ought not to be able to override a `private` or `internal` method, which might lead to an object security violation.

Other solutions such as signaling an ambiguity error when encountering the expression `c.f()` or alternately even preventing class `C` from being defined are also incorrect for the same reasons as above.

The only sensible thing to do is to define the language so that the expression `c.f()` in this case returns `"fB"`. The rule for looking up an unqualified property *p* of an object *o* is therefore:

- First, find the highest (least derived) class that defines a property of *o* named *p* that's visible in the currently `use`d namespaces; let *n* be that member's namespace (if there is more than one such namespace in the same class, just pick one).
- Second, find and return the lowest (most derived) definition of *o*`.`*n*`::`*p*.

The rule for looking up a qualified property *n*`::`*p* of an object *o* is the same as in object-oriented programming — return the lowest (most derived) definition of *o*`.`*n*`::`*p*. Thus, in the example above `c.public::f()` will return `"fC"` regardless of the `use namespace` declarations in effect.

Perhaps not coincidentally, defining the rules this way makes JavaScript 2.0 much easier to compile by allowing partial evaluation of property lookups in many common cases.

8 Attributes and Conditional Compilation

Several of the previous sections implicitly referenced attributes. This section explores them in a little more detail and uncovers an interesting and perhaps unexpected use of attributes for conditional compilation.

In JavaScript 2.0, attributes are simple constant expressions that can be listed in front of most definitions as well as a few other statements. Examples of attributes already mentioned include `static`, `public`, `private`, `explicit`, `dynamic`, `final`, as well as namespaces. These are, in fact, constant expressions, and other attributes may be defined using `const` definitions. Multiple attributes may be listed on the same definition. Attaching multiple namespaces to a definition simultaneously defines several qualified names, one for each namespace, which is valuable for many versioning scenarios.

Attributes may be placed before the opening braces of a block, which distributes the attributes among all the definitions inside that block. Thus, instead of

```
class A {
  static private const x;
  static private const y;
  static private const z;
  function f() {}
}
```

one can write:

```
class A {
  static private {
    const x;
    const y;
    const z;
  }
  function f() {}
}
```

Two other very useful attributes are `true` and `false`. The attribute `true` is ignored. `false` causes the entire definition or statement to disappear without being evaluated or processed further. By defining a global boolean constant (or obtaining one from the environment) one can achieve convenient conditional compilation without resorting to a separate preprocessor language. In the example below, instances of class `A` have the slot `count` and method `g` only if the `const` definition of `debug` is changed to `true`:

```
const debug = false;

class A {
  var x;
  debug var count;

  function f() {}
  debug function g() {}
}
```

9 Conclusion

This paper presents only the highlights and some of the rationale of JavaScript 2.0.

There are a few other features, such as units, typed arrays, and operator overriding that were not covered here. See [7] for a much more detailed description.

It's been a long road to get to this point, with various proposals being discussed in the ECMA TC39 working group for several years. Most of what made the problem so difficult and time-consuming is the difficult task of retaining backward compatibility. JavaScript 1.5 has grown many ad-hoc features that don't carry forward well and tend to make any future evolution even more ad-hoc and especially complicated. Trying to avoid complexity while retaining compatibility has been a constant struggle.

Netscape's implementations [8] of JavaScript 1.5 and the forthcoming 2.0 are available as open source under the NPL, GPL, or LGPL license. The JavaScript source code is compact and stand-alone and does not depend on the rest of the browser to be compiled; it's been embedded in hundreds if not thousands of different environments to date.

Bibliography

[1] Martin Abadi, Luca Cardelli. *A Theory of Objects*. Springer-Verlag 1996.

[2] ECMA web site, `http://www.ecma.ch/`

[3] ECMA-262 Edition 1 standard,
`http://www.mozilla.org/js/language/E262.pdf`

[4] ECMA-262 Edition 2 standard,
`http://www.mozilla.org/js/language/E262-2.pdf`

[5] ECMA-262 Edition 3 standard,
`http://www.mozilla.org/js/language/E262-3.pdf`

[6] David Flanagan. *JavaScript: The Definitive Guide*, 4th Edition. O'Reilly 2002.

[7] Mozilla's JavaScript 2.0 web site,
`http://www.mozilla.org/js/language/js20/`

[8] Mozilla's JavaScript web site, `http://www.mozilla.org/js/`

[9] Netscape. *Object Hierarchy and Inheritance in JavaScript*. Thunder Lizard Productions JavaScript Conference '98, also available at
`http://developer.netscape.com/docs/manuals/communicator/jsobj/index.htm`

[10] SmallScript LLC. SmallScript Website, `http://www.smallscript.net/`

[11] W3C Document Object Model, `http://www.w3.org/DOM/`