Published: 12 June 2020

RESEARCH-ARTICLE

JavaScript: the first 20 years

ALLEN WIRFS-BROCK

BRENDAN EICH, Brave Software, Inc., San Francisco, CA, United States

JavaScript: The First 20 Years

ALLEN WIRFS-BROCK, Wirfs-Brock Associates, Inc., USA
BRENDAN EICH, Brave Software, Inc., USA

Shepherds: Sukyoung Ryu, KAIST, South Korea
Richard Gabriel (poet, writer, computer scientist), California

How a sidekick scripting language for Java, created at Netscape in a ten-day hack, ships first as a de facto Web standard and eventually becomes the world's most widely used programming language. This paper tells the story of the creation, design, evolution, and standardization of the JavaScript language over the period of 1995–2015. But the story is not only about the technical details of the language. It is also the story of how people and organizations competed and collaborated to shape the JavaScript language which dominates the Web of 2020.

CCS Concepts: • **General and reference → Computing standards, RFCs and guidelines**; • **Information systems → World Wide Web**; • **Social and professional topics → History of computing**; **History of programming languages**; • **Software and its engineering → General programming languages**; *Scripting languages.*

Additional Key Words and Phrases: JavaScript, ECMAScript, Standards, Web browsers, Browser game theory, History of programming languages

ACM Reference Format:
Allen Wirfs-Brock and Brendan Eich. 2020. JavaScript: The First 20 Years. *Proc. ACM Program. Lang.* 4, HOPL, Article 77 (June 2020), 189 pages. https://doi.org/10.1145/3386327

Contents

Authors' addresses: Allen Wirfs-Brock, allen@wirfs-brock.com, Wirfs-Brock Associates, Inc., Sherwood, Oregon, USA; Brendan Eich, brendan@brave.com, Brave Software, Inc., San Francisco, California, USA.

2475-1421/2020/6-ART77
https://doi.org/10.1145/3386327

1 INTRODUCTION

In 2020, the World Wide Web is ubiquitous with over a billion websites accessible from billions of Web-connected devices. Each of those devices runs a Web browser or similar program which is able to process and display pages from those sites. The majority of those pages embed or load source code written in the JavaScript programming language. In 2020, JavaScript is arguably the world's most broadly deployed programming language. According to a Stack Overflow [2018] survey it is

used by 71.5% of professional developers making it the world's most widely used programming language.

This paper primarily tells the story of the creation, design, and evolution of the JavaScript language over the period of 1995–2015. But the story is not only about the technical details of the language. It is also the story of how people and organizations competed and collaborated to shape the JavaScript language which dominates the Web of 2020.

This is a long and complicated story. To make it more approachable, this paper is divided into four major parts—each of which covers a major phase of JavaScript's development and evolution. Between each of the parts there is a short interlude that provides context on how software developers were reacting to and using JavaScript.

In 1995, the Web and Web browsers were new technologies bursting onto the world, and Netscape Communications Corporation was leading Web browser development. JavaScript was initially designed and implemented in May 1995 at Netscape by Brendan Eich, one of the authors of this paper. It was intended to be a simple, easy to use, *dynamic language*[g] that enabled snippets of code to be included in the definitions of Web pages. The code snippets were interpreted by a browser as it rendered the page, enabling the page to dynamically customize its presentation and respond to user interactions.

Part 1, The Origins of JavaScript, is about the creation and early evolution of JavaScript. It examines the motivations and trade-offs that went into the development of the first version of the JavaScript language at Netscape. Because of its name, JavaScript is often confused with the *Java*[g] programming language. Part 1 explains the process of naming the language, the envisioned relationship between the two languages, and what happened instead. It includes an overview of the original features of the language and the design decisions that motivated them. Part 1 also traces the early evolution of the language through its first few years at Netscape and other companies.

A cornerstone of the Web is that it is based upon non-proprietary open technologies.[1] Anybody should be able to create a Web page that can be hosted by a variety of Web servers from different vendors and accessed by a variety of browsers. A common specification facilitates interoperability among independent implementations. From its earliest days it was understood that JavaScript would need some form of standard specification. Within its first year Web developers were encountering interoperability issues between Netscape's JavaScript and Microsoft's reverse-engineered implementation. In 1996, the standardization process for JavaScript was begun under the auspices of the Ecma International standards organization. The first official standard specification for the language was issued in 1997 under the name "ECMAScript." Two additional revised and enhanced editions, largely based upon Netscape's evolution of the language, were issued by the end of 1999.

Part 2, Creating a Standard, examines how the JavaScript standardization effort was initiated, how the specifications were created, who contributed to the effort, and how decisions were made.

By the year 2000, JavaScript was widely used on the Web but Netscape was in rapid decline and Eich had moved on to other projects. Who would lead the evolution of JavaScript into the future? In the absence of either a corporate or individual "Benevolent Dictator for Life,"[2] the responsibility for evolving JavaScript fell upon the ECMAScript standards committee. This transfer of design responsibility did not go smoothly. There was a decade-long period of false starts, standardization hiatuses, and misdirected efforts as the committee tried to find its own path forward evolving the language. All the while, usage of JavaScript rapidly grew, often using implementation-specific

[1] The specifications of Web technologies are not developed and controlled by a single company and any company or organization may create and distribute implementations of the technologies that interoperate with other implementations.
[2] A technology evolution approach where a single person or organization is recognized as the permanent sole authority able to make decisions to modify or extend a technology. Some projects (particularly open source ones) grant this authority to the individual who started the project or first developed the technology. [Meyer 2014]

extensions. This created a huge legacy of unmaintained JavaScript-dependent Web pages and revealed new interoperability issues. Web developers began to create complex client-side JavaScript Web applications and were asking for standardized language enhancements to support them.

Part 3, Failed Reformations, examines the unsuccessful attempts to revise the language, the resulting turmoil within the standards committee, and how that turmoil was ultimately resolved.

In 2008 the standards committee restored harmonious operations and was able to create a modestly enhanced edition of the standard that was published in 2009. With that success, the standards committee was finally ready to successfully undertake the task of compatibly modernizing the language. Over the course of seven years the committee developed major enhancements to the language and its specification. The result, known as ECMAScript 2015, is the foundation for the ongoing evolution of JavaScript. After completion of the 2015 release, the committee again modified its processes to enable faster incremental releases and now regularly completes revisions on a yearly schedule.

Part 4, Modernizing JavaScript, is the story of the people and processes that were used to create both the 2009 and 2015 editions of the ECMAScript standard. It covers the goals for each edition and how they addressed evolving needs of the JavaScript development community. This part examines the significant foundational changes made to the language in each edition and important new features that were added to the language.

Wherever possible, the source materials for this paper are contemporaneous primary documents. Fortunately, these exist in abundance. The authors have ensured that nearly all of the primary documents are freely and easily accessible on the Web from reliable archives using URLs included in the references. The primary document sources were supplemented with interviews and personal communications with some of the people who were directly involved in the story. Both authors were significant participants in many events covered by this paper. Their recollections are treated similarly to those of the third-party informants.

The complete twenty-year story of JavaScript is long and so is this paper. It involves hundreds of distinct events and dozens of individuals and organizations. Appendices A through E are provided to help the reader navigate these details. Appendices A and B provide annotated lists of the people and organizations that appear in the story. Appendix C is a glossary that includes terms which are unique to JavaScript or used with meanings that may be different from common usage within the computing community in 2020 or whose meaning might change or become unfamiliar for future readers. The first use within this paper of a glossary term is usually italicized and marked with a "g" superscript like this: "*term*[g]." Appendix D defines abbreviations that a reader will encounter. Appendix E contains four detailed timelines of events, one for each of the four parts of the paper.

1.1 Names, Numbers, and Abbreviations

The world of JavaScript can be a confusing place with multiple names for what is seemingly the same thing. This is exacerbated when simultaneously entering the world of standard-setting organizations, which often use two and three letter abbreviations and numbers to identify their organizational units and work products. In order to minimize this confusion we will start by defining some of these names and abbreviations and set some conventions that are used throughout the rest of this paper.

"JavaScript" is the common name of the programming language that was originally developed by Netscape Communications Corporation for use in Web pages. Its uses, both on the Web and in other environments, have grown far beyond that and every day millions of programmers think and talk about this language using that name. The JavaScript programming language is distinct and technically very different from the Java programming language but the similarity of their names is a frequent source of confusion.

Abbreviation	Edition	Date	Project Editors	Pages
ES1	1st	June 1997	Guy Steele	95
ES2	2nd	August 1998	Mike Cowlishaw	101
ES3	3rd	December 1999	Mike Cowlishaw	172
ES3.1	5th	Internal working name for 5th edition		
$ES4_1$ and $ES4_2$	4th	Abandoned, never completed		
ES5	5th	December 2009	Pratap Lakshman Allen Wirfs-Brock	245
ES5.1	5.1	June 2011	Allen Wirfs-Brock	245
ES6 or ES2015	6th	June 2015	Allen Wirfs-Brock	545
ES2016	7th	June 2016	Brian Terlson	546

Fig. 1. ECMA-262 Editions, 1997–2016

JavaScript® is also a registered trademark. The trademark was originally registered by Sun Microsystems, and as of the date of this paper the registration is owned by Oracle Corporation. The trademark was licensed by Sun to Netscape and later to the Mozilla Foundation. Netscape and Mozilla have used names such as "JavaScript 1.4" to describe specific versions of their implementations of the language. Some implementors of the language have used other names in order to avoid possible trademark issues. Because of the multiple names, the trademark issues, and the confusion with Java many contemporary users, book authors, and tool implementors simply call the language "JS" and "js" is commonly used as a file extension for JavaScript source code. Within this paper, we use the unqualified term "JavaScript" when we are generically talking about the language and its usage outside the context of a specific version, host environment, or implementation.

The word "JavaScript" was avoided when the standard specification was created for the language; instead, that specification uses the name "ECMAScript." The names "JavaScript" and "ECMAScript" are essentially different names for the same thing. In this paper when we use the term "ECMAScript" we are specifically talking about the language as defined by the standard.

The "ECMA" part of ECMAScript is derived from Ecma International, the Swiss-based standards organization under whose auspices the ECMAScript standards are developed. "ECMA" was originally an acronym for "European Computer Manufacturers Association," the original name of the organization that evolved into Ecma International. That organization no longer considers "Ecma" to be an acronym. Ecma International currently capitalizes only the "E" in "Ecma" but at various times in the past they have used all capital letters. That was the case when ECMAScript was first developed and the reason that the language name starts with five capital letters. In this paper, we will usually use the word "Ecma" when referring to the Ecma International standards organization.

Ecma develops many computing-related standards. The actual work of developing standards occurs within Ecma Technical Committees, abbreviated as "TC." When a new Ecma TC is created, it is assigned a serial number to uniquely identify it. TC39 is the TC that was created to standardize JavaScript. Some Ecma TCs are subdivided into Task Groups, abbreviated as "TG," with specific responsibilities. From 2000 through 2007, TC39's responsibility was expanded to include other programming languages in addition to JavaScript. During that period, responsibility for ECMAScript was assigned to TC39-TG1. In this paper, we use "TG1" as the abbreviation for TC39-TG1.

Ecma assigns a number to each distinct standard authored by its TCs and uses those numbers prefixed with "ECMA-" as a designators The ECMAScript standard is designated "ECMA-262." When a standard is revised, a new edition is issued using the same number suffixed with an edition number. For example, the third version of the ECMAScript standard was officially known as "ECMA-262,

3rd Edition." Informally within TC39, and ultimately within the broader JavaScript community the convention emerged of using an abbreviation like "ES3" as a shorthand for the official edition designation, the "ES" standing for "ECMAScript." Figure 1 lists the editions of *ECMA-262*[g] along with the abbreviations used in this paper.

The attempt to define a 4th edition spanned nearly ten years and consisted of two largely independent design efforts. In this paper, the terms "$ES4_1$" and "$ES4_2$" are used to refer specifically to one or the other of those efforts.[3] "ES4" is used to refer to the overall effort to create a 4th edition.

Starting with publication of the 6th edition, TC39 adopted the convention of using the year of publication as the abbreviation. So both "ES6" and "ES2015" are informal abbreviations for "ECMA-262, 6th Edition" but "ES2015" is preferred. However, "ES6" was the most commonly used abbreviation during the development of the 6th edition. TC39 members also used "Harmony[g]" and "ES.next[g]" as code-names to refer to the 6th edition development project.

This paper uses numerous in-line code snippets to illustrate JavaScript concepts. Some of the snippets are valid only for specific versions or editions of JavaScript/ECMAScript. Other snippets illustrate proposed features that never became part of the language. Throughout the paper, snippets which are not valid for all versions of JavaScript/ECMAScript are appropriately labeled.

Part 1: The Origins of JavaScript

2 PREHISTORY

The concept and foundation technologies of the World Wide Web were developed during 1989–1991 by Tim Berners-Lee [2003] at CERN. Berners-Lee's Web technologies circulated within the high-energy physics community for a couple of years. However, they did not receive much attention outside that community until Marc Andreessen, an undergraduate student, and Eric Bina, working at the University of Illinois at Urbana-Champaign National Center for Supercomputing Applications (NCSA), developed *Mosaic*[g] in 1992–1993.

NCSA Mosaic was an easy to install, easy to use Web client with a graphic user interface. It essentially defined the software category "Web browser" and popularized the concept of the World Wide Web outside of the physics community. Mosaic was widely distributed and by early 1994 commercial interests were scrambling to get on the browser bandwagon by either licensing the NCSA Mosaic code or by building Mosaic-inspired browsers from scratch. Jim Clark, the founder of Silicon Graphics Inc., obtained venture capital funding and recruited Marc Andreessen and Eric Bina. In April 1994 they co-founded the company that would eventually be named Netscape Communications Corporation. Netscape set as its goal replacing NCSA Mosaic as the world's most popular browser. It developed from scratch an enhanced next generation Mosaic-like browser that it started widely distributing in October 1994. By early 1995 *Netscape Navigator*[g] had achieved its initial goal and was rapidly displacing Mosaic.

Tim Berners-Lee's Web technology was centered around using the *declarative*[g] HTML markup language to describe documents for presentation as Web pages. In contrast there was considerable industry interest in using *scripting languages*[g] [Ousterhout 1997] to enable end users to orchestrate the operation of their applications. Languages such as Visual Basic in Microsoft Office and AppleScript [Cook 2007] are not intended for implementing the complex data structures and algorithmic components that exist at the core of major applications. Instead, they provide a way for users to glue together such application components in novel ways. As Netscape expanded the audience for the World Wide Web, an important question was if and how scripting should integrate into Web pages.

[3] TC39 and the members of the two ES4 design efforts did not use the "$ES4_1$" or "$ES4_2$" nomenclature. They simply referred to their then-current work as "ES4."

2.1 Brendan Eich Joins Netscape

Brendan Eich,[4] in 1985, completed his masters degree at the University of Illinois Urbana-Champaign and immediately went to work for Silicon Graphics, Inc. He worked primarily on the Unix kernel and networking layers. In 1992 he left SGI to join MicroUnity, a well-funded startup developing video media processors. At both companies he implemented small special-purpose languages that supported kernel and networking programming tasks. At MicroUnity he also did some work on the GCC *compiler*[g].

In early 1995, Brendan Eich was recruited to Netscape with the bait of "come and do Scheme in the browser."[5] But when Eich joined Netscape on April 3, 1995, he found a complex product marketing and programming language situation. Netscape had rebuffed a low-priced acquisition offer from Microsoft in late 1994, after which Netscape management expected a direct attack via Microsoft's "Embrace, Extend, Extinguish" strategy [Wikipedia 2019]. Microsoft, under Bill Gates' direct leadership, had quickly realized that its forthcoming proprietary walled-garden information utility, Project Blackbird [Anderson 2007], would be irrelevant with the rise of the Web as a cross-OS platform. Gates' "Internet Tidal Wave" memo [Gates 1995] rebooted Microsoft from Blackbird to *Internet Explorer*[g] and a full suite of server products, as Netscape rushed to stake claims in the same markets.

The candidates for a Web page scripting language included research languages such as Scheme; practical Unix-based languages such as Perl, Python, and Tcl; and proprietary languages such as Microsoft's Visual Basic. Brendan Eich was expecting to implement Scheme in the browser. But in early 1995 Sun Microsystems had started a guerrilla marketing campaign [Byous 1998] for its still unreleased[6] Java language. Sun and Netscape quickly engaged with each other to strike a deal whereby Java would be integrated into Netscape 2. Eich recalls that the rallying cry articulated by Marc Andreessen at Netscape meetings was "Netscape plus Java kills Windows." On May 23, 1995, at Sun's public announcement of Java, Netscape announced its intent to license Sun's Java technology [Netscape 1995a] for use in the browser.

Rapid strategizing inside Netscape to choose a scripting language severely handicapped Scheme, Perl, Python, Tcl, and Visual Basic as not viable due to business interests and/or time to market considerations. The only approach considered viable by senior managers at Netscape and Sun, notably Marc Andreessen and Sun's Bill Joy, was to design and implement a "little language"[7] to complement Java.

Doubters, dominant at Sun and a majority at Netscape, questioned the need for a simpler scripting language: wasn't Java suitable for scripting; would it be possible to explain why two languages were better than one; and did Netscape have the necessary expertise to create a new language.

The first objection was easily countered. Java in spring 1995 was not a suitable language for beginners. One had to wrap a main program's code body in a static *method*[g] named `main` in a *class*[g] declaration in a package. One had to declare static *types*[g] for all parameters, return values, and variables. Based on experience with Visual BASIC complementing Visual C++, and many Unix languages complementing native-code-based components, it was clear Java was not simple enough for the "glue" scripters.

The second objection was overcome by citing Microsoft's products. For professional Windows application programmers, Microsoft sold Visual C++. For amateurs, part-time programmers, designers, accountants, and others, Microsoft provided Visual Basic as the scripting language by which those

[4] The book *Coders At Work* [Seibel 2009, chapter 4] includes a more detailed look at Eich's early career.
[5] Referring to the Scheme programming language [Sussman and Steele Jr 1975].
[6] The stealth alpha release of Java was in March/April 1995.
[7] Jon Bentley [1986] introduced the term "little language" to characterize a small easy-to-learn language that is "specialized to a particular problem domain and does not include many features found in conventional languages."

less-experienced, part-time programmers could "glue" together and customize components built using Visual C++. A version of Visual Basic called "Visual Basic for Applications" was integrated into the Microsoft Office applications to support user extension and scripting of those applications.

Having overcome the first two objections, Marc Andreessen proposed the code-name "Mocha" for the browser scripting language with, according to Eich, the hope that the language would be renamed "JavaScript" in due course. This companion language to Java would have to "look like Java" while remaining easy to use and "object-based" rather than class-based, like Java.

That still left a final remaining objection: did Netscape have the expertise to create an effective scripting language and have it ready for the Netscape 2 beta in September 1995. Brendan Eich's assignment was to prove that it did by creating Mocha.

2.2 The Story of Mocha

With the Java announcements imminent, Brendan Eich saw time as of the essence and a bird in the hand worth many hypotheticals in bushes; and so he prototyped the first *Mocha*[g] implementation in ten contiguous days in May, 1995.[8] This work was rushed to meet a feasibility demonstration deadline. The demo consisted of the bare minimum language implemented and minimally integrated into the Netscape 2 pre-alpha browser.

Eich's prototype was developed on a Silicon Graphics Indy Unix workstation [Netfreak 2019]. The prototype used a hand-written lexer and recursive-descent parser. The parser emitted bytecoded instructions rather than a parse tree. The bytecode *interpreter*[g] was simple and slow.[9]

Bytecode was a requirement of Netscape's LiveWire server[10] whose developers were counting on embedding Mocha even before it was prototyped. The team's ex-Borland management and engineering staff were big believers in dynamic scripting languages but wanted bytecodes, rather than source parsing, for faster server application loading.

Marc Andreessen stressed that Mocha should be so easy to use that anyone could write a few lines directly within an HTML document. Upper management at Sun and Netscape reiterated the requirement that Mocha "look like Java," explicitly ruling out anything like BASIC. But the Java-like appearance created an expectation of Java-like behavior that impacted both the design of the *object*[g] model and the semantics of "primitive types" such as `boolean`, `int`, `double`, and `string`.

Other than looking like Java, Brendan Eich was free to select most language design details. After joining Netscape, he had explored "easy to use" or "pedagogical" languages, including HyperTalk [Apple Computer 1988], Logo [Papert 1980], and Self [Ungar and Smith 1987]. Everyone agreed that Mocha would be "object-based," but without classes, because supporting classes would take too long and risk competing with Java. Out of admiration of Self, Eich chose to start with a dynamic object model using *delegation*[g] with a single prototype link. He believed that would save implementation time, although in the end he lacked sufficient time to expose that mechanism in the Mocha prototype.

Objects are created by applying the `new` operator to a *constructor function*[g]. A default object constructor function named `Object` is built into the environment along with other built-in objects. Each object is composed of zero or more properties. Each *property*[g] has a name (also called a *property key*[g]) and a value which can be either a *function*[g], an object, or a value of one of several other built-in data types. Properties are created by assigning a value to an unused property key. There are no visibility or assignment restrictions for properties. A constructor function may provide

[8] There is no known record of the specific dates but Brendan Eich believes it was May 6–15.

[9] It used a large *discriminated union*[g] to represent the different types of data *values*[g] and used reference counting for memory management.

[10] Brendan Eich had spent his first month at Netscape officially working in the server group.

an initial set of properties; additional properties can be added to an object after its creation. This very dynamic approach was especially favored by the LiveWire team.

Although the lure of Scheme was gone, Brendan Eich still found Lisp-like *first-class*[g] functions attractive. Without classes to contain methods, first-class functions provided a toolkit for Scheme-inspired idioms: top-level procedures, passing functions as arguments, methods on objects, and event handlers. The time constraints required deferral of function expressions (also called *lambda expressions*[g], or just lambdas) but they were reserved in the grammar. Event handlers and object methods were unified by borrowing the `this` keyword from Java (after C++) in any function to denote the contextual object on which that function was invoked as a method.

Motivated by informal discussions with Marc Andreessen and a few early Netscape engineers[11] the prototype supported an `eval` function that could parse and execute a string containing a program. The intuition was that this kind of dynamic string-to-program programming would be important for some applications on Web browsers and servers.[12] But the decision to support `eval` had immediate consequences. Some uses required functions to provide their source code as a string, via a Java-like `toString` method. Eich chose to implement a bytecode decompiler in his ten-day sprint[13], because source code primary storage or recovery from secondary storage seemed too costly for some required target architectures. This was especially the case for Windows 3.1 personal computers that were constrained by the Intel 8086 16-bit segmented memory model, requiring overlays and manually managed multi-segment memory for unbounded or large in-memory structures.

At the end of the ten days, the prototype was demonstrated (Figure 2) at a meeting of the full Netscape engineering staff. It was a success, which led to excessive optimism about shipping a more complete and fully integrated version for the Netscape 2 release whose first beta release was scheduled for September. Brendan Eich's primary focus for the summer was to more fully integrate Mocha into the browser. This would require designing and implementing the APIs that enabled Mocha programs to interact with Web pages. At the same time he had to turn the language's prototype implementation into shippable software and respond to early internal users' bug reports, change suggestions, and feature requests.

More details of the 10-day creation of Mocha are in Brendan Eich's retellings of this story [Eich 2008c, 2011d; JavaScript Jabber 2014; Walker 2018]. The source code of the production version of Mocha is available via the Internet Archive [Netscape 1997b]. Jamie Zawinski's [1999] "the netscape dorm" is a contemporaneous account of the experience of working for Netscape as a software developer during this period.

3 JAVASCRIPT 1.0 AND 1.1

Netscape Communications Corporation and Sun Microsystems announced JavaScript on December 4, 1995, in a joint press release [Netscape and Sun 1995; Appendix F]. The press release describes JavaScript as "an object scripting language" that would be used to write scripts that dynamically "modify the properties and behaviors of Java objects." It would serve as a "complement to Java for easy online application development." The companies were attempting to forge a strong brand linkage between the Java and JavaScript languages even though their technical designs were only superficially similar. The name similarity and its implication that the languages are closely related has been a continuing source of confusion.

[11] Including John Giannandrea who had worked for General Magic where two programming languages were built that could be used both client- and server-side.

[12] For example, to enable a form of partial evaluation or to support server execution of client-provided code, similar to Telescript [General Magic 1995] agents.

[13] Developers in 1995 would not have used the term "sprint." However it is a good characterization of Eich's effort.

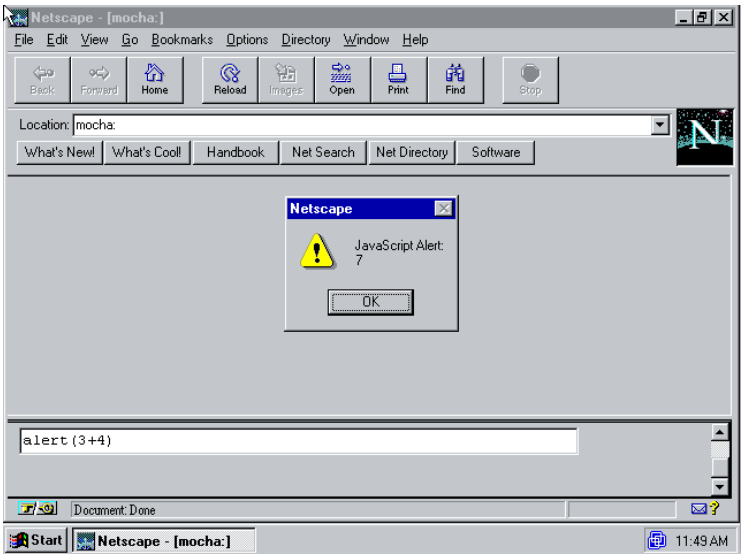


Fig. 2. The Mocha Console. Brendan Eich's initial demo of Mocha featured a "Mocha Console" running in a pre-alpha version of Netscape 2 on a SGI Unix workstation. The same Mocha Console shipped, essentially unchanged except for its name, as part of the production release of Netscape 2. This is a screen capture of Netscape 2.02 running on Windows 95. The Mocha console was activated by typing `mocha:` into the browser address bar—for production Netscape 2 this was changed to `javascript:` but `mocha:` still worked. Activating the console caused a two-frame page to open in the browser. Mocha expressions typed into the text box of the lower frame were evaluated for effect in the context of the upper frame. This example shows the built-in `alert` function being called to display a popup containing the computed value of an expression. The original demo version would have displayed "Mocha Alert" in the popup instead of "JavaScript Alert."

JavaScript, under the name "LiveScript," was initially exposed to the public in September 1995 as part of the first beta release [Netscape 1995b] of Netscape Navigator 2.0. That release was followed by four more beta releases leading up to the March 1996 production release of Navigator 2.0, which supported JavaScript 1.0. Netscape Enterprise Server 2.0 also shipped in March [Netscape 1996f] and incorporated JavaScript 1.0 within its LiveWire server-side scripting component.

JavaScript was only one relatively minor feature of Netscape Navigator. As such, its development was constrained by the overall Navigator 2.0 schedule that required a feature freeze in August 1995. The JavaScript 1.0 feature set was essentially a triage of what was working or near working in the Mocha implementation that August. The feature set was incomplete relative to the envisioned language design and exhibited various problematic bugs and edge case behaviors even though Eich continued to fix bugs in the initial Mocha implementation throughout the Navigator 2.0 release process. Interviewed [Shah 1996] shortly before the 1.0 release, Brendan Eich echoed the official positioning of JavaScript as an adjunct to Java and the rushed nature of the initial release:

> BE[Brendan Eich]: I hope it [JavaScript] will be implemented by other vendors, based on the spec that Bill Joy and I are working on. I'd like to see it remain small, but become ubiquitous on the web as the favored way of gluing HTML elements and actions on them together with Java applets and other components.
> BE: …For all I know, the most common use is to make pages a little smarter and more live—for instance, make a click on a link load a different *URL*[s] depending on the time of day.

> ...
> BE: There is light at the end of the tunnel, although because JavaScript was too much of a one-man show, [Netscape] 2.0 will contain numerous annoying little bugs. My hope is that all big bugs have workarounds, and I've spent a lot of time working with developers to find bugs and workarounds.
>
> I'm following through for 2.1 by fixing bugs, adding features, and trying to make JavaScript consistent across all our platforms. I don't know when 2.1 will ship, but would wager it'll be out well before next fall—we move fast here.

JavaScript 1.0 [Netscape 1996d] was a simple *dynamically typed*[g] language supporting numeric, string, and Boolean values; first-class functions; and, an object data type. Syntactically, JavaScript, like Java, was in the C family with control flow statements borrowed from C and an expression syntax that included most of the C numeric operators. JavaScript 1.0 had a small library of built-in functions. JavaScript 1.0 source code was usually directly embedded in HTML files, but the built-in library included an `eval` function that could parse and evaluate JavaScript source code encoded as a JavaScript string value. JavaScript 1.0 was a very lean language. Figure 3 is a summary of some of the absent features whose omission is likely surprising to modern JavaScript programmers.

In early 1996, work on began on "Atlas" [Netscape 1996g], the code name for what would ship as Netscape Navigator 3.0 in August 1996. Brendan Eich was able to resume work on features that were incomplete or missing at the August 1995 2.0 feature freeze. It was only with the release of JavaScript 1.1 [Netscape 1996a,e] in Navigator 3.0 that the initial definition and development of JavaScript was completed. The following sections present an overview of the design of the JavaScript 1.0/1.1 language.

3.1 JavaScript Syntax

The syntax of JavaScript 1.0 was directly modeled after the statement syntax of the C programming language [ANSI X3 1989] with some *AWK*[g] [Aho et al. 1988] inspired embellishments. A script is a sequence of statements and declarations. Unlike C, JavaScript statements are not restricted to occurring within the body of a function. In JavaScript 1.0, source code for a script is embedded within HTML documents surrounded by a `<script></script>` tag.

The C-inspired statements in JavaScript 1.0 are the expression statement; the `if` conditional statement; the `for` and `while` iteration statements; the `break`, `continue`, and `return` statements for non-sequential flow control; and the statement block which enables a `{}`-delimited sequence of statements to be used as if it were a single statement. The `if`, `for`, and `while` statements are compound statements.[14] JavaScript 1.0 did not include C's `do-while` statement, `switch` statement, statement labels, or `goto` statement.

To the basic suite of C statements, JavaScript 1.0 added two compound statements for accessing the properties of its object data type. The AWK inspired `for-in` statement iterates over the *property keys*[g] of an object. Within the body of a `with` statement[15] the properties of a designated object can be accessed as if their names were declared variables. Because properties may be dynamically added (and in later versions of the language deleted) the visible variable *bindings*[g] may change as execution progresses within a `with` statement's body.

JavaScript declarations do not follow the style of C or Java declarations. JavaScript is dynamically typed; moreover, it does not have language-level type names to serve as syntactic prefixes for

[14] A compound statement contains nested statements as part of its syntactic structure. Typically a statement block is used as a nested statement. Most kinds of compound statements have a single nested statement. In that case, the nested statement is the "body" of the compound statement.

[15] The `with` statement was added after the ten-day Mocha sprint at the request of the Netscape LiveWire team.

A distinct `Array` object type	`Array` literals
Regular expressions	Object literals
A global binding for `undefined`	`===` operator
`typeof`, `void`, `delete` operators	`in`, `instanceof` operators
`do-while` statement	`switch` statement
`try-catch-finally` statement	`break`/`continue` to label
Nested function declarations	Function expressions
Function `call` and `apply` methods	`prototype` property of functions
Prototype-based inheritance	Access to built-in prototype objects
Cyclic garbage collection[g]	HTML `<script>` tag `src` attribute

Fig. 3. Commonly used JavaScript features (circa 2010) not present in JavaScript 1.0

recognizing declarations. Instead, JavaScript declarations are keyword prefixed. JavaScript 1.0 has two forms of declarations: `function` declarations and `var` declarations. The syntax of `function` declarations[16] was directly borrowed from AWK. A `function` declaration defines the name, formal parameters, and statement body of a single callable function. A `var` declaration introduces one or more variable bindings and optionally assigns values to the variables. All `var` declarations are treated as statements and may occur in any statement context, including within block statements. In JavaScript 1.0/1.1 `function` declarations may occur only at the top level of a script and may not contain nested `function` declarations. A `var` declaration may occur within a function body and the variables defined by such declarations are local to the function.

Unlike C, JavaScript 1.0 statement blocks do not introduce declaration scopes. Within a function body, `var` declarations within a block are locally visible to the entire function body. A `var` declaration within a block outside of a function has global *scope*[g]. Assignment to a variable name that does not have an in-scope `function` or `var` declaration implicitly creates a global variable with that name. This behavior has proven to be a significant source of errors as mistyping the name of a declared variable silently creates a new variable with the mistyped name.

One major departure from traditional C syntax is JavaScript's treatment of semicolons at the end of statements. While C treats semicolons as a mandatory statement terminator, JavaScript allows statement-terminating semicolons to be left out when they are the last significant character on a line. The exact rules for this behavior were not included in the JavaScript 1.0 documentation. The Netscape 2.0 Handbook does not show semicolons when describing the various JavaScript statement forms. It simply says: "A single statement may span multiple lines. Multiple statements may occur on a single line if each statement is separated by a semi-colon [Netscape 1996d]." A semicolon-free coding style was the norm used in the Handbook's JavaScript code examples such as the following:

```
var a, x, y
var r=10
with (Math) {
  a = PI * r * r
  x = r * cos(PI)
  y = r * sin(PI/2)
}
```

The ability to write JavaScript code without using semicolons is known as Automatic Semicolon Insertion (ASI). ASI remains controversial among JavaScript programmers; a significant fraction

[16]Including the syntax and semantics of the `return` statement.

of programmers still prefer to write code in a semicolon-free style and others would prefer that nobody ever used ASI.

3.2 Data Types and Expressions

JavaScript 1.0/1.1 is a dynamically typed language with five fundamental data types: number, string, Boolean, object, and function. By "dynamically typed" we mean that runtime type information is associated with each datum rather than value containers such as variables. Runtime type checks ensure that operations are applied only to data values which are supported by each operation.

Booleans, strings, and numbers are immutable values. The Boolean type has two values, named `true` and `false`. String values consist of immutable sequences of 8-bit character codes. There is no support for Unicode. The number type consists of all possible IEEE 754 [IEEE 2008] double-precision binary 64-bit floating-point values with the exception that only a single canonical NaN value is exposed. Some operations give special treatment to number values that correspond to unsigned 32-bit integers and signed 32-bit 2's complement integers. Mocha internally used an alternative representation for such integer values but officially there was only a single numeric data type.

JavaScript 1.0 has two special values that represent the absence of a useful data value. Uninitialized variables are set to the special value *undefined*.[17] This is also the value returned when a program attempts to access the value of a non-existent property of an object. In JavaScript 1.0 the value *undefined* may be accessible by declaring and accessing an uninitialized variable. The value `null` is intended to represent "no object" in contexts where an object value is expected. It is modeled after Java's `null` value and it facilitates integration of JavaScript with objects implemented using Java. Throughout its entire history the existence of these two similar but observably different values has caused confusion among JavaScript programmers many of whom are uncertain about when they should use one of them instead of the other.

JavaScript 1.0's expression syntax is copied from C with generally the same set of operators and precedence rules. The major omissions are C's pointer and type-related operators and the unary + operator. The binary + operator is overloaded to perform both numeric addition and string concatenation. The shift and bit-wise logical operators operate upon the bit level encoding of signed 32-bit 2's complement integers. If necessary, operands are truncated to integers and modulo reduced to 32-bit values. The >> operator performs a sign-extending arithmetic right shift of a 32-bit integer value. JavaScript adds the >>> operator, borrowed from Java, which performs an unsigned right shift.

JavaScript 1.1 adds the `delete`, `typeof`, and `void` operators. In JavaScript 1.1 the `delete` operator simply sets its variable or object-property operand to the value `null`. The `typeof` operator returns a string identifying the primitive type of its operand. Its possible string values are `"undefined"`, `"object"`, `"function"`, `"boolean"`, `"string"`, `"number"`, or an implementation-defined string value that identifies a kind of host-defined object. Surprisingly, `typeof null` returns the string value `"object"` rather than `"null"`. This is arguably consistent with Java where all values are objects and `null` is essentially the "no object" object. However, Java lacks an equivalent to the `typeof` operator and uses `null` as the default value of uninitialized variables. Brendan Eich's recollection is that the value of `typeof null` was the result of a *leaky abstraction*[g] in the original Mocha implementation. The runtime value of `null` was encoded with the same internal tag value used for object values and hence the implementation of the `typeof` operator returned `"object"` without needing any extra special-case logic. This choice has proven to be a great annoyance to JavaScript programmers who typically want to test if a value is actually an object before attempting to use the value as the base for accessing a property. But testing that `typeof` a value is `"object"`

[17] We italicize "undefined" in this section because JavaScript 1.0 did not provide a name for directly accessing this value.

From type	To type: function	object	number	boolean	string
undefined	error	null	error	false	`"undefined"`
function	N/C	Function object	valueOf/error	valueOf/true	decompile
object		N/C			
(not null)	Function object		valueOf/error	valueOf/true	toString/valueOf[1]
(null)	error		0	false	`"null"`
number			N/C		
(zero)	error	null		false	`"0"`
(nonzero)	error	Number		true	default*
(NaN)	error	Number		false[2]	`"NaN"`
(+Infinity)	error	Number		true	`"+Infinity"`
(-Infinity)	error	Number		true	`"-Infinity"`
boolean		Boolean		N/C	
(false)	error		0		`"false"`
(true)	error		1		`"true"`
string		String			N/C
(empty)	error		error[3]	false	
(non-empty)	error		number/error	true	

Key:
When two results separated by a slash, JavaScript tries the first, and if unsuccessful, uses the second.
N/C: No Conversion Necessary.
decompile: A string containing the function's canonical source.
toString: The result of calling the toString method.
valueOf: The result of calling the valueOf method, if it returns a value of the To type.
number: Numeric value if string is a valid integer or floating-point literal.
[1] If valueOf does not return a string, the default object-to-string conversion is used.
[2] JavaScript 1.1 as implemented in Navigator 3.0 converts NaN to true.
[3] JavaScript 1.1 as implemented in Navigator 3.0 converts the empty string to 0.

Fig. 4. JavaScript 1.1 Type Coercions as presented by Eich and McKinney [1996, page 23] in their preliminary JavaScript 1.1 specification. The type coercions rules that were eventually standardized are slightly different. This is a facsimile of the original table with minor typographical differences. Footnote 3 did not appear in the original.

is an insufficient guard for a property access because attempting to access a property of `null` produces a runtime error.

The `void` operator simply evaluates its operand and then returns *undefined*. An idiom for accessing *undefined* is `void 0`. The void operator was introduced as an aid in defining HTML hyperlinks that execute JavaScript code when clicked, for example:

```
<a href="javascript:void usefulFunction()">Click to do something useful</a>
```

The value of an `href` *attribute*[g] should be a URL and `javascript:` is a special URL protocol that is recognized by browsers. It means evaluate what follows as JavaScript code and use the result, converted to a string, as if it was the response-document fetched using a normal `href` URL. The `<a>` element will attempt to process that response document unless it is *undefined*. Usually a Web developer wants the JavaScript expression to be evaluated only for its effects when the link is clicked. Prefixing an expression with `void` permits it to be used in that manner and avoids further processing by the `<a>` element.

The most significant difference between C and JavaScript expressions is that JavaScript operators automatically coerce their operands to data types in the domain of the operators. JavaScript 1.1

```
//using Object constructor
var pt = new Object;
pt.x=0;
pt.y=0;
```

```
//using custom constructor
function Point(x,y) {
  this.x = x;
  this.y = y;
}
var pt = new Point(0,0);
```

Fig. 5. JavaScript 1.0 Object Creation Alternatives. Properties can be added to an object after it is created by the `Object` or added during creation by using a custom constructor function.

added a configurable mechanism for coercing arbitrary objects to number or string values. Figure 4 summarizes the JavaScript 1.1 coercion rules.

3.3 Objects

JavaScript 1.0 objects are associative arrays whose elements are called "properties." Each property has a string key and a value, which may be any JavaScript data type. Properties may be dynamically added. JavaScript 1.0/1.1 does not provide any way to remove a property from an object.

Properties whose key strings conform to the syntax rules for identifiers may be accessed using a dot notation, for example `obj.prop0`. All properties, including those whose keys are not identifiers may be accessed using a bracket notation where the brackets surround an expression that is evaluated and converted to a string that is used as the property key. For example `obj["prop"+n]` is equivalent to `obj.prop0` when the value of `n` is `0`. Assigning to a non-existent property creates a new property. Accessing the value of a non-existent property usually returns the value *undefined*. However, in JavaScript 1.0/1.1 the value `null` is returned if a non-existent property value is accessed using bracket notation and the property key is the string representation of a non-negative integer.

Properties may be used both as data stores and to associate behavior with objects. A property whose value is a function may be invoked as a method of the object. Functions invoked as methods of an object have access to the object via the dynamic binding of the keyword `this` (§3.7.4).

Objects are created by applying the `new` operator to a built-in or user-defined function. A function that is intended to be used in this manner is called a "constructor." Constructors typically add properties to the new object. The properties may be either data stores or methods. The built-in constructor `Object` may be used to create a new object that initially has no properties. Figure 5 shows how either the `Object` constructor or a user-defined constructor function can be used to create new objects.

JavaScript 1.0 also has a built-in `Array` constructor but the only observable difference between an object created using the `Object` constructor and the `Array` constructor is the debug string displayed for the object. Objects created by the JavaScript 1.0 `Array` constructor do not have a `length` property.

Array-like indexing behavior can be achieved for any object by creating properties using integer values as the property keys. Such an object may also have properties with non-integer keys:

```
var a = new Object;  //or new Array
a[0] = "zero";
a[1] = "one";
a[2] = "two";
a.length = 3;
```

```
//define functions to be used as methods
function ptSum(pt2) {return new Point(this.x+pt2.x, this.y+pt2.y)}
function ptDistance(pt2) {
  return Math.sqrt(Math.pow(pt2.x - this.x, 2) + Math.pow(pt2.y - this.y,2));
}

//define Point constructor
function Point(x,y) {
  //create and initialize a new object's data properties
  this.x = x;
  this.y = y;
  //add methods to each instance object
  this.sum = ptSum;
  this.distance = ptDistance;
}
var origin = new Point(0,0); //create a Point object
```

Fig. 6. Defining a Point abstraction using JavaScript 1.0. Each instance object has its own method properties.

JavaScript 1.0 has no concept of object *inheritance*[g]. Programs must individually add all properties to each new object. This is typically done by defining a constructor function for each "class" of object used by the program. Figure 6 shows the definition of a simple Point abstraction written using JavaScript 1.0. The important things to note in this example are as follows:

- Each method must be defined as a globally visible function. Such functions must be given names which are unlikely to conflict with the names used to define the method functions of other class-like abstractions (`ptSum`, `ptDistance`).
- When an object is constructed, an object property must be created for each method with its value initialized to the corresponding global function.
- Methods are invoked using their property name (`origin.distance`) rather than their declared global name (`ptDistance`).

JavaScript 1.1 eliminates the need to create method properties directly on each new instance. It associates a *prototype*[g] object with each constructor function via a property, named `prototype`, of the function object. The 1.1 *JavaScript Guide* [Netscape 1996e] describes `prototype` as "a property that is shared by all objects of the specified type." This is a vague description that might have been better stated as: an object whose properties are shared with all objects created by a constructor.

The sharing mechanism is not further described but it is possible to observe the following characteristics of prototype objects:

- Accessing a property of an object whose property name is defined on the prototype associated with the object's constructor returns the value of the prototype object's property.
- Adding or modifying a property of a prototype object is immediately visible to already existing objects created by the constructor associated with the prototype.
- Assignment of a property value to an object *shadows*[g][18] the value of an identically named property defined on the prototype associated with the object's constructor function.

Each property of the built-in `Object.prototype` object is visible via property access on any object unless the property has been shadowed by the object or its prototype.

[18]Creates a new property that over-rides access to the prototype's property.

```
//define functions to be used as methods
function ptSum(pt2) {return new Point(this.x+pt2.x, this.y+pt2.y)}
function ptDistance(pt2) {
  return Math.sqrt(Math.pow(pt2.x - this.x, 2) + Math.pow(pt2.y - this.y,2));
}

//define Point constructor
function Point(x,y) {
  //create/initialize a new object's data properties
  this.x = x;
  this.y = y;
}
//add methods to shared prototype object
Point.prototype.sum = ptSum;
Point.prototype.distance = ptDistance;

var origin = new Point(0,0); //create a Point object
```

Fig. 7. Defining a Point abstraction using JavaScript 1.1. Instance objects inherit method properties from the `Point.prototype` object rather than defining method properties on each instance.

Figure 7 shows the JavaScript 1.1 the definition of the simple Point abstraction from Figure 6. It differs in that the methods are installed only once on the prototype object rather than repeatedly during construction of each instance object. A property provided to an object by a prototype property is called an *inherited property*[g]. A property defined directly on an object is called an *own property*[g]. An own property shadows an identically-named–own property.

The properties of a prototype object are usually methods. In that case, the prototype provided by a constructor is serving the same role as a C++ vtable or Smalltalk MethodDictionary—it associates common behaviors with a set of objects. The constructor is essentially serving the role of a class object and its prototype is the container of the methods which are shared by instances of the class. This is a reasonable interpretation of JavaScript 1.1's object model but not the only one.

The naming of the constructor `prototype` property is a clear hint that Brendan Eich had another object model in mind. That model was inspired by the Self programming language [Ungar and Smith 1987]. In Self, a new object is created by partially cloning the prototypical object of some category of objects. Each clone has a `parent` link back to the prototype so that the prototype can provide the features intended to be common to all of its clones. The JavaScript 1.1 object model can be viewed as a variant of the Self model where the prototype objects are accessed indirectly via constructor functions and the `new` operator clones new instances from the prototype. The cloned instances *inherit*[g] the properties of the prototype objects as common shared features. Some JavaScript programmers call this mechanism "prototypal inheritance[g]." It is a form of delegation. Some JavaScript programmers also use the double entendre "classical inheritance[g]" to refer to the style of inheritance used in Java and many other object-oriented languages.

The JavaScript 1.1 documentation [Netscape 1996e] does not fully describe either of these object models. It maintained a marketing story consistent with the December 1995 Netscape/Sun press release. JavaScript was positioned as a language for scripting object interactions while the actual definition of object abstractions (class definitions) were to be written in Java. Native JavaScript object abstraction capabilities were limited to secondary features that drew minimal attention and were largely undocumented.

3.4 Function Objects

In JavaScript 1.0/1.1, a function definition creates and names a callable function. JavaScript functions are first-class object values. The name provided in a `function` declaration defines a global variable, similar to a `var` declaration in top-level code. Its value is the function object and may be assigned to variables, set as property values, passed as arguments in function calls, and returned as values from functions. Because functions are objects they may have properties defined on them. The following examples shows how a property can be added to a function object:

```
function countedHello() {
  alert("Hello, World!");
  countedHello.callCount++; //increment this function's callCount property
}
countedHello.callCount = 0; //associate counter with function and initialize
for (var i=0; i<5; i++) countedHello();
alert(countedHello.callCount);  //displays: 5
```

Functions are declared with a formal parameter list, but the size of the parameter list does not limit the number of arguments that can be passed when calling the function. If a function is called with fewer arguments than its declared number of parameters, the extra parameters are set to *undefined*. If a function is called with more arguments than the number of formal parameters, the extra arguments are evaluated but their values are not available via parameter names. However, an array-like arguments object is available as the value of the function object's `arguments` property during execution of the body of the function. All of the actual arguments passed in a call to the function are available as integer-keyed properties of the `arguments` object. This enables a function to be written that can process a variable length arguments list.

3.5 Built-in Library

JavaScript 1.0 comes with a library of built-in functions, objects, and constructors. The library defines a small number of general-purpose objects[19] and functions along with a larger set of host-specific objects and functions. For Netscape Navigator, *host objects*[g] provided a model of portions of the current HTML document. These APIs ultimately became known as the Document Object Model (DOM) level 0 [Koch 2003; Netscape 1996b]. For Netscape Enterprise Server, host objects supported client/server communications, managing the state of client and server sessions, and file and database access. That design for server host objects did not achieve adoption beyond Netscape server products.

The early design of JavaScript was largely driven by the needs of the browser platform. The Netscape documentation for the early JavaScript versions did not explicitly distinguish between library elements that were intended to be host environment independent or host dependent. However, the design, evolution, and standardization of the DOM and other browser platform APIs constitute its own significant story deserving its own history. The current paper mentions browser related issues only when they are relevant to the overall design of JavaScript.

JavaScript 1.0 has only two general-purpose object Classes: `String` and `Date`. In addition, there is a singleton global object `Math`, whose properties are commonly used mathematical constants and functions. The constructors for several inactive or incompletely implemented Classes are also observable to JavaScript 1.0 programs that know how to access them. JavaScript 1.1 completes

[19] Lack of a formally named object abstraction mechanism makes it difficult to talk about specific kinds of objects supported by the JavaScript library. JavaScript documentation has used various terms including "type," "object," "constructor," and "class" to talk about such abstractions. In the remainder of this paper we use the capitalized word "Class" when we need to talk about the definition of a set of JavaScript objects that share a common representation and methods, regardless of the actual form of the definition.

Base Objects 1.0	Base Objects 1.1	Properties 1.0	Properties Added in 1.1
<global functions>		eval, isNaN,[1] parseFloat,[2] parseInt[2]	
~~Array~~[3]	Array		join, reverse, sort, toString
~~Boolean~~[3]	Boolean		toString
Date		getDate, getDay, getHours, getMinutes, getMonth, getSeconds, GetTime, getTimezoneOffset, getYear, setDate, setHours, setMinutes, setMonth, setSeconds, setTime, setYear, toGMTString, toLocaleString, Date.parse, Date.UTC	toString
<function objects>		arguments, length, caller	
~~Function~~[3]	Function		prototype, toString
Math		E, LN2, LN10, LOG2E, LOG10E, PI, SQRT1_2, SQRT2, abs, acos, asin, atan, ceil, cos, exp, floor, log, max, min, pow, random,[1] round, sin, sqrt, tan	
Object			constructor, eval, toString, valueOf
~~Number~~[3]	Number		toString, Number.NaN, Number.MAX_VALUE, Number.MIN_VALUE, Number. NEGATIVE_INFINITY, Number.POSITIVE_INFINITY
<string values>		length	
	String	charAt,[4] indexOf, lastIndexOf, ~~split~~[3], substring, toLowerCase, toUpperCase, (plus 13 HTML wrapper methods)	split, toString, valueOf

[1] In 1.0 available only on Unix platforms.
[2] In 1.0 behavior differs depending upon host operating system.
[3] Exists in 1.0 but is not operational or buggy.
[4] In 1.0 these methods appear to be properties of string values. In 1.1 they are properties of String.prototype.

Fig. 8. JavaScript 1.0/1.1 Host-Independent Built-In Library

the implementation of these features and documents their existence. Figure 8 summarizes the host-independent Classes defined in JavaScript 1.0 and 1.1.

The `String` Class provides the `length` property and six general-purpose methods that operate upon immutable string values and, when appropriate, return new string values. The JavaScript 1.0 `String` Class also includes thirteen methods for wrapping a string value with various HTML tags. This is an example of the fluid boundary between host-dependent and general-purpose functionality in JavaScript 1.0/1.1. JavaScript 1.0 does not provide a global `String` constructor function. All string values are created using string literals or by operators and built-in functions. JavaScript 1.1 adds the global `String` constructor and the `split` method.

The `Date` Class is used to represent calendar dates and time. JavaScript 1.0 `Date` was a direct transliteration, bugs and all, of the `java.util.Date` class of Java 1.0 [Gosling et al. 1996]. This includes encoding details such as using a millisecond resolution time value centered on 00:00:00 GMT on January 1, 1970, externally numbering months from 0–11, and Year 2000 ambiguities that were present in the Java design. This design choice was motivated by Java interoperability

requirements. The only Java methods excluded were `equals`, `before`, and `after` which were not needed because JavaScript's automatic coercions permitted its numeric relational operators to be directly used with Date objects.

Other than `Object`, `Date` is the only usable built-in constructor function in JavaScript 1.0. `Date` is also the only Class that exposed methods on the constructor object in addition to methods for Class instances. None of the browser-specific Classes exposed a constructor function.

Some properties of built-in library objects and host-provided objects have characteristics which are not available for properties defined by JavaScript programmers. For example, their method properties are not enumerated by the `for-in` statement. Some of their properties are ignored by the `delete` operator or have read-only values. Accessing or modifying some of their properties trigger special behaviors with observable side effects.

JavaScript 1.1 adds a usable `Array` Class. The `Array` constructor creates objects intended for use as integer-index, zero-origin vectors of heterogeneous values. The array elements are presented as object properties whose keys are the string representation of their integer indices. Array objects also have a `length` property whose value is initially set by the constructor. The `length` property's value is updated whenever an element index that is greater or equal to the current `length` value is accessed. Thus the number of elements of an Array object may dynamically grow.

3.6 Execution Model

In Netscape 2 and subsequent browsers, an HTML Web page may contain multiple `<script>` elements. When a page is loaded, a fresh JavaScript execution environment and global context is created for the HTML document. The global context includes its global object, which is an object whose property keys are the names of the built-in functions and variables provided by JavaScript and the host environment plus the global variables and functions defined by the scripts.

In Netscape 2, the JavaScript code for each `<script>` element is parsed and evaluated in the order they occur within the page's HTML file. In later browsers `<script>` elements may be tagged for deferred evaluation which lets the browser continue processing HTML while it waits for the JavaScript code to be retrieved from the network. In either case the browser evaluates one script at a time. Scripts normally share the same global object. Global variables and functions created by a script are visible to all subsequent scripts. Each script is run to completion without preëmption or interruption. This characteristic of early browsers became a fundamental principle of JavaScript. Scripts are atomic units of execution and once started each one runs until it is completed. Within a script it is not necessary to worry about interference from concurrent execution of other scripts because it cannot occur.

Netscape 2 also introduced the concept of Web page frames.[20] A frame is a region of a Web page into which a separate HTML document may be loaded. All of the frames on a page share the same JavaScript execution environment but each frame has a separate global context within that environment. Scripts loaded in different frames see a different global object, different built-ins, and different global variables and functions. But a global context is not an address space. A JavaScript execution environment has a single address space of objects that is shared among all of the frames within that environment. Because of this single address space of objects, it is possible for object references to be passed among the JavaScript code in different frames intermingling objects from different global contexts. This can lead to surprising behavior. Consider the JavaScript 1.1 example in Figure 9. Each frame has its own distinct `Object` constructor and `Object.prototype` that provide properties inherited by all objects created by that constructor. Adding a property to

[20]The original HTML `<frame>` tag is considered obsolete and has been superseded by the `<iframe>` tag. The semantics described in this section are common to both kinds of elements.

```
//The variable alien references an object created within a different frame
//by evaluating: new Object()
var alien = createNewObjectInADifferentFrame();
var native = new Object(); // create an object in the current frame
Object.prototype.sharedProperty = "each frame has distinct built-ins";
alert(native.sharedProperty);  //displays: each frame has distinct built-ins
alert(alien.sharedProperty);   //displays: undefined
```

Fig. 9. JavaScript 1.1 example showing that code in different HTML frames can interchange objects even though they have distinct built-in objects.

a frame's `Object.prototype` does not make the property visible to objects created by another frame's `Object` constructor.

Interactive JavaScript Web pages are event-driven applications where the event loop is provided by the browser. HyperCard [Apple Computer 1988] inspired Brendan Eich to use the concept of events in the original Netscape 2 DOM [Netscape 1996c] design. Originally events were triggered primarily by user interactions, but in modern browsers there are many kinds of events, only some of which are user originated.

When all the scripts defined by a Web page have been executed, the JavaScript environment for the page remains active waiting for an event to occur. Event handlers can be associated with objects provided by the browser, including many DOM objects. An event handler is simply a JavaScript function that is called in response to the occurrence of an event. Assigning a function to certain properties of browser objects makes that function the handler for the event associated with the property. For example, objects that correspond to clickable pointing devices have an `onclick` property that can be set. A JavaScript event handler can also be defined directly in an HTML element using a snippet of JavaScript code, for example:

```
<button onclick="doSomethingWhenClicked()">Click me</button>
```

When the HTML element is processed the browser creates a JavaScript function and assigns it as the value of the `onclick` property of the button object. The `onclick` code snippet is used as the function body. When an event with a JavaScript event handler occurs it is placed into a pool of pending events. When no JavaScript code is executing, the browser takes a pending event from the event pool and calls the associated function. Like scripts, event handler functions run to completion.

3.7 Oddities and Bugs

JavaScript has several unusual or unexpected features. Some were intentional and others were artifacts of quick design decisions made during the original Mocha 10-day sprint. JavaScript 1.0 also had bugs and incompletely implemented features.

3.7.1 Redundant Declarations. JavaScript tolerates multiple declarations of the same name within a scope. All declarations of a name within a function correspond to a single binding that is visible throughout the entire body of the function. For example, the following is a valid function definition:

```
function f(x, x) {      // x names the second parameter, ignores 1st x
  var x;                // same binding as second parameter
  for (var x in obj) { // same binding as second parameter
    var x=1, x=2;       // same bindings as second parameter
  }
  var x=3;              // same binding as second parameter
}
```

All of the var declarations within the function f refer to the same variable binding which is also the binding of the second parameter of the function. The same name can occur more than once in a function's formal parameter list. Prior to executing the function body, all variables defined by var declarations are initialized to *undefined* except for var variables whose names are also parameter names. In that case, the initial value is the same as the argument passed for the identically named parameter. The initializers of var declarations, including redundant declarations, have the same semantics as an assignment to the initialized variable. They are executed when reached in the normal sequence of execution within the function body.

There may be multiple function declarations with the same name in a script. When this occurs it is the last function declaration for the name that is hoisted to the top of the script and used to initialize the global variable with that name. Any other function declarations for that name are ignored. If there are both global function declarations and global var declarations for the same name they all refer to the same variable and any var declarations with an initializer will overwrite the function value if and when the initializer is encountered during the sequence of execution.

3.7.2 Automatic Coercions and the == Operator. Automatic coercions were intended to lower the entry barrier for the initial adoption of JavaScript as a simple scripting language. However, as JavaScript evolved into a general purpose language the coercions have proven to be a significant source of confusion and coding bugs. This is particularly true for the == operator. Some of the problematic coercions added to Mocha after the initial ten-day sprint were in response to alpha user requests to ease the integration of JavaScript and HTTP/HTML. For example, internal Netscape users requested that HTTP status codes containing the string value "404" should compare equal to the number 404 using == comparison. They also requested automatic coercion of empty strings to 0 in numeric contexts, providing a default value for empty fields of HTML forms. These coercions introduced surprises such as: 1 == '1' and 1 == '1.0' but '1' != '1.0'.

JavaScript 1.0 treats the = operator as == within the predicate of an if statement, for example:

```
if (a = 0) alert("true");   //these two statements are equivalent
if (a == 0) alert("true");
```

JavaScript 1.0–1.2

3.7.3 32-Bit Arithmetic. JavaScript's bitwise logical operators operate on 32-bit values encoded within an IEEE double. The bitwise operators first integer truncate and then do a modulo conversion of their operands to 32-bit 2's complement values before performing the bitwise operation. So, a Number value, x, can be forced to a 32-bit value by the expression x|0 where | is the bitwise logical or operator. Using this idiom 32-bit signed addition can be performed as follows:

```
function int32bitAdd(x, y) {
  return ((x|0) + (y|0))|0 //addition with result truncated to 32-bits
}
```

Unsigned 32-bit arithmetic can be performed using a similar pattern but using the unsigned right shift operator >>>0 instead of |0.

3.7.4 The this *keyword.* Every function has an implicit this parameter. When a function is called as a method, the this parameter is set to the object that was used to access the method. This is the same meaning that is given to this (or alternatively self) in most object-oriented languages. However, JavaScript's use of a single form of definition for both object-associated methods and standalone functions has made this a source of confusion and bugs for many programmers.

When a function is directly called, without being qualified with an object, this is implicitly set to the global object. The properties of the global object include all of a program's global variables,

so this qualified property references in a directly called function are equivalent to global variable references. Because the treatment of this depends upon how a function is called, the same this reference can have different meanings during different calls, for example:

```
function setX(value) {this.x=value}
var obj = new Object;
obj.setX = setX;  //install setX as a method of obj

obj.setX(42);     //calls setX as a method
alert(obj.x);     //displays: 42

setX(84);         //directly call setX
alert(x);         //accesses global variable x; displays 84
alert(obj.x);     //displays: 42
```

Further confusion about this arises because some HTML constructs implicitly turn JavaScript code fragments into functions that are invoked as methods, for example in:

```
<button name="B" onclick="alert(this.name + " clicked")>Click me</button>
```

when the event handler is executed, it invokes the onclick method of the button; this refers to the button object and this.name retrieves the value of its name attribute.

3.7.5 Arguments Objects. A function's arguments object is joined to its formal parameters—there is a dynamic mapping between the arguments object's numerically indexed properties and the function's formal parameters. A change to an arguments object property also changes the value of the corresponding formal parameter, and a change to a formal parameter is observable as a change to the corresponding arguments object property:

```
f(1,2);
function f(argA, argB) {
  alert(argA);  // displays: 1
  alert(f.arguments[0]);  // displays: 1
  f.arguments[0] = "one";
  alert(argA);            // displays: one
  argB = "two";
  alert(f.arguments[1]);  // displays: two
  alert(f.arguments.argB);// displays: two
}
```

JavaScript 1.0–1.1

As shown in the last line of the above example, the formal parameters can also be accessed by using their names as property keys of the arguments object.

Conceptually, when a function is called, a new arguments object is created for the new activation of the function and the value of the function object's arguments property is set to that new arguments object. But in JavaScript 1.0/1.1 the function object and the arguments object are the same objects:

```
function f(a,b) {
   if (f==f.arguments) alert("f and f.arguments are the same object")
}
if (f.arguments==null) alert("but only while a call to f is active")
```

JavaScript 1.0–1.1

Ideally, a function's arguments object should be accessible only within its body. This is partially enforced by automatically setting a function's arguments property to null when the function

returns from a call. But assume there are two functions, `f1` and `f2`. If `f1` calls `f2` then `f2` can access the arguments of `f1` by evaluating `f1.arguments`.

An `arguments` object also has a property named `caller`. The value of the `caller` property is the function object that invoked the current activation of the function or `null` if it is the outermost function activation. By using `caller` and `arguments`, any function can inspect the functions and their arguments on the current call stack and even modify the formal parameter values of functions on the call stack. A `caller` property with the same meaning is also directly accessible via a function object without going through its `arguments` object.

3.7.6 Special Treatment of Numeric Property Keys. In JavaScript 1.0 the bracket notation has an unusual semantics when used with integer keys. In some cases, a bracketed integer key will access an object's properties in their creation order. A property order access occurs with an integer if a property with that key does not already exist on the object and the value, n, of the integer is less than the total number of object properties. In that case, the n^{th} property (zero-origined) that was created on that object is accessed, for example:

```
var a = new Object;  //or new Array
a[0] = "zero";
a[1] = "one";
a.p1 = "two";

alert(a[2]);  // displays: two
a[2] = "2";
alert(a.p1);  // displays: 2
```

JavaScript 1.0

JavaScript 1.1 removed this special treatment of bracket notation.

3.7.7 Properties of Primitive Values. In JavaScript 1.0 numbers and Boolean values do not have properties, and attempting to access or assign a property to them produces an error message. String values behave as if they are objects with properties but they all share the same set of properties and values except for their read-only `length` property, for example:

```
"xyz".prop = 42;     // Set the value of property prop to 42 for all strings
alert("xyz".prop);   // displays: 42
alert("abc".prop);   // displays: 42
```

JavaScript 1.0

In JavaScript 1.1 property access or assignments to a number, Boolean, or string value causes a "wrapper object" to be implicitly created using the built-in Number, Boolean, or String constructors. The property access is performed upon the wrapper and typically accesses an inherited property from its built-in prototype. Coercions performed by automatically invoking `valueOf` and `toString` methods permit wrappers to be used as if they were primitive values in most situations. It is possible to create a new property on a wrapper object by assignment, but implicitly created wrappers typically become inaccessible immediately after the assignment, for example:

```
"xyz".prop = 42;     // Set the value of a String wrapper property to 42
alert("xyz".prop);   // Implicitly creates another wrapper, displays: undefined
var abc = new String("abc");  // Explicitly create a wrapper object
alert(abc+"xyz");    // Implicitly converts wrapper to string, displays: abcxyz
abc.prop = 42;       //create a property on a wrapper objects
alert(abc.prop);     // display: 42
```

JavaScript 1.1

3.7.8 HTML Comments inside JavaScript. A potential interoperability problem with JavaScript in Netscape 2 was caused by what Netscape 1 and Mosaic browsers did when they encountered an HTML `<script>` element. Those older, but still widely used browsers, would display the `<script>` body—the actual JavaScript source code—as text when they displayed a Web page. This could be prevented in those browsers by enclosing the script body with an HTML comment,[21] for example:

```
<script>
  <!-- This is an HTML comment surrounding a script body
    alert("this is a message from JavaScript"); //not visible to old browsers
    //the following line ends the HTML comment
  -->
</script>
```

Mosaic and Netscape 1

Using this coding pattern, the HTML parsers in Netscape 1 and Mosaic would recognize the entire script body as an HTML comment and not display it. But, as originally implemented in Mocha this would prevent the browser from parsing (and executing) the script body as JavaScript because the HTML comment delimiters were not syntactically valid in JavaScript code. To circumvent that problem, Brendan Eich made JavaScript 1.0 accept `<!--` as the start of a single line comment, equivalent to `//`. He did not make `-->` a recognized JavaScript comment deliminator because putting a `//` in front of it would suffice for this pattern. A backward interoperable script could then be written as follows:

```
<script>
  <!-- This is an HTML comment in old browsers and a JS single line comment
  alert("this is a message from JavaScript"); //not visible to old browsers
  //the following line ends the HTML comment and is a JS single line comment
  //-->
</script>
```

Mosaic, Netscape 1, and Netscape 2 with JavaScript 1.0

Even though `<!--` comments were not documented as official JavaScript syntax, they were used by Web developers and supported by other browser JavaScript implementations. The result, was that `<!--` became part of the de facto *Web Reality*[g]. It took twenty years, but in 2015 they were added to the ECMAScript standard—eventually Web Reality always wins.

4 MICROSOFT JSCRIPT[22]

The same week Netscape and Sun publicly announced JavaScript, Microsoft announced that it intended to make Visual Basic "a standard for creating World Wide Web-based applications using Visual Basic Script" [Wingfield 1995]. Microsoft formally announced support for JavaScript in its May 29, 1996, Internet Explorer 3.0 Beta Press Release [Microsoft 1996]:

> **ActiveX Script.** With native support for Visual Basic® Script and JavaScript, Microsoft Internet Explorer 3.0 provides the most comprehensive and language-independent script capabilities. Microsoft Internet Explorer can be extended to support additional scripting languages such as REXX, CGI and PERL. Web page designers can plug any scripting language into their HTML code to create interactive pages that link together ActiveX controls, Java Applets and other software components.

[21] As long as the script body does not contain any > or -- operators, which are illegal in HTML comments.

[22] Most of the material in this section is based upon a recorded interview Allen Wirfs-Brock conducted on March 22, 2018, with Robert Welland, Shon Katzenberger, and Peter Kukol [Welland et al. 2018].

Work on what became JScript started in October 1995 when Robert Welland joined Microsoft's Internet Explorer (IE) team. Welland had previously worked for Apple on the Newton handheld computers and the NewtonScript language [Smith 1995]. NewtonScript was a prototype-based object-oriented language whose design was influenced by the Self language. Welland had worked closely with Walter Smith who was the principal designer of NewtonScript and with David Ungar who had been a consultant to the project so Welland was very familiar with Self and Ungar's ideas about prototype-based languages. After Apple Welland had been thinking about how scripting could be added to browsers. This led to him being hired to put scripting into Internet Explorer.

When Robert Welland got to Microsoft he was told he should put Visual Basic into IE but when he talked to the Visual Basic team in Microsoft's Developer Tools Division (*DevDiv*[g]) they said it would take two years. So he and Sam McKelvie quickly did the work to get Visual Basic for Applications[23] running within IE 2 but found it was too complicated to integrate with the browser's object model. Welland observed LiveScript/JavaScript in the Netscape 2 public betas and started experimenting with a simple bytecode interpreter for JavaScript which McKelvie then improved. Welland discovered that Peter Kukol in DevDiv had written a JavaScript parser[24] that could generate bytecodes. Welland and McKelvie connected their interpreter with Kukol's parser and a garbage collector written by Patrick Dussud to form the foundation of JScript.

Microsoft's DevDiv was responsible for the development of all of Microsoft's programming languages and developer tools so the involvement of Robert Welland and Sam McKelvie, who worked for the IE team in the Windows division, in the development of a new language implementation was politically sensitive. There was also internal controversy about whether IE should support JavaScript. DevDiv wanted to focus its attention on Visual Basic for scripting and on Java for applications but the IE team's goal was for IE 3 to be compatible with Netscape 3 and that required including JavaScript support. Microsoft was not happy about having to support JavaScript but it was too late to ignore it. The compromise was that IE and Microsoft as a whole would support both JavaScript and Visual Basic for scripting and that responsibility for scripting languages would belong to DevDiv. The IE/Windows team would be responsible for integrating scripting into the browser and other products.

In January 1996, Sam McKelvie transferred into DevDiv while Robert Welland remained on the IE team. Also in January, Shon Katzenberger transferred into DevDiv from the Microsoft Word team to work on scripting. Katzenberger took over responsibility for the interpreter and, with help from the Visual Basic team, got a scripting subset of Visual Basic running on the same interpreter. This became known as Visual Basic Script or VBS.

Welland and McKelvie packaged the scripting system, including support for both JScript and VBS, as an embeddable component that became known as Active Scripting. This component shipped in 1996 as part of both IE3 and Microsoft's Web server product, IIS, where it provided the server-side scripting technology for Active Server Pages. Active Scripting subsequently became a standard component of Microsoft Windows and as of 2019 was still available to support legacy applications.

The IE team was very focused on competing with Netscape. They hoped that the script debugger that was part of Active Scripting would attract JavaScript Web developers to IE because Netscape did not have a JavaScript debugger. But they also understood that website interoperability with Netscape was going to be essential to the adoption of IE. Shon Katzenberger and others ran developmental versions of IE 3 against thousands of websites that used JavaScript and compared the results with Netscape 2 and Netscape 3. Whenever they found a difference, Katzenberger had

[23] Visual Basic for Applications is a variant of Visual Basic 6 that is embedded within Microsoft Office applications.
[24] When interviewed in 2018, Kukol recounted that he had recently visited the JavaScript team at Microsoft and discovered that his original parser was still used (with extensions) by Microsoft's then-current JavaScript implementation.

to reverse engineer the Netscape JavaScript behavior to understand what it was doing differently. Some of the behaviors they found came as great surprises. They were particularly shocked when they discovered that in Netscape's implementation HTML frames shared a common object address space and could freely interchange objects. IE had implemented frames as isolated environments and it took significant reëngineering to enable objects to be passed among them.

Throughout the entire JScript development process, the lack of a proper language specification was a constant problem. Welland recalled that during its development Thomas Reardon, who led the overall IE3 development effort, took every opportunity he had to chide his counterparts at Netscape about the lack of a JavaScript language specification.

5 FROM MOCHA TO SPIDERMONKEY

For all of 1995 and most of 1996 Brendan Eich was the the only Netscape developer working full-time on the *JavaScript engine*[g].[25] JavaScript 1.1 in the August 1996 production release of Netscape 3.0 still consisted primarily of code from the 10-day May 1995 prototype. After this release, Eich felt it was time to pay down the technical debt[26] of the *engine*[g] and work at making JavaScript "a cleaner language." Netscape management wanted him to work on a language specification. They were sensitive to criticism from Microsoft about the lack of a specification and were anticipating that the imminent start of standardization activities would require a specification as input. Eich resisted. He wanted to start by reimplementing Mocha. To write a specification he would have to carefully review the Mocha implementation. He thought it would be most efficient to rewrite Mocha as he reviewed it. That would also enable him to correct original design mistakes before enshrining them in a specification.

Frustrated with this debate, Brendan Eich left the office and worked from home for two weeks during which he redesigned and reimplemented the core of the JavaScript engine. The result was a faster, more reliable, and more flexible execution engine. He discarded representing JavaScript values as *discriminated unions*[g] and used tagged pointers containing immediate primitive values instead. He implemented features such as nested functions, function expressions, and a switch statement that never made it into the original engine. The reference counting memory manager was replaced with a mark/sweep garbage collector.

When Eich returned to the office, the new engine replaced Mocha. Chris Houck, one of the original Netscape developers, joined Eich as the second full-time member of the JavaScript team. Houck named the new engine "*SpiderMonkey*[g]"[27] based upon a lewd line from the movie *Beavis and Butt-Head Do America* [Judge et al. 1996]. Clayton Lewis joined the team as manager and hired Norris Boyd. Rand McKinny, a technical writer, was assigned to assist Eich in writing a specification.

Brendan Eich continued to enhance the language as JavaScript 1.2, for release as part of Netscape 4.0. Its first beta release was in December 1996. Regular expressions were added in the April 1997 beta. Production releases of Netscape 4 for various platforms started in June and were spread over the second half of 1997.

The JavaScript 1.2 language and built-in library implemented by SpiderMonkey were significantly enhanced relative to JavaScript 1.0/1.1. Figure 10 lists the major new features in JavaScript

[25] In the JavaScript community, the term "engine" refers to a JavaScript language implementation. A JavaScript engine typically consists of parser, a virtual machine or similar runtime support, a garbage collector, a standard library implementation, and other components.

[26] This is Brendan Eich's retrospective description. "Technical debt" is not a term he would have used in 1996 to describe the need to catch up with deferred maintenance.

[27] SpiderMonkey became the name of the JavaScript subsystem of subsequent Netscape and Mozilla browsers. As of 2020, Mozilla still uses that name even though the actual implementation technology has changed multiple times.

- do statement
- statement labels and `break`/`continue` to label
- `switch` statement
- Nested function declarations (lexical scoping)
- Function expressions (lambda expressions)
- Eliminate automatic coercions previously performed by == operator
- Property delete operator actually deletes properties
- Object literals
- Array literals
- Regular Expression literals
- `RegExp` objects with methods to do regular expression matching
- `__proto__` pseudo property of all objects
- New Array methods: `push`, `pop`, `shift`, `unshift`, `splice`, `concat`, `slice`
- New String methods: `charCodeAt`,
- `fromCharCode` (ISO latin-1), `match`, `replace`, `search`, `substr`, `split` using `RegExp`
- function `arity` property
- A function and its `arguments` object are distinct objects
- A function's formal parameters and local declarations are accessible as named properties of its `arguments` object
- `arguments.callee`
- `watch`/`unwatch` functions
- `import`/`export` statements and signed scripts

Fig. 10. New Feature In JavaScript 1.2

1.2 [Netscape 1997c]. Most of the library additions were inspired by features available in other popular languages. The Array `concat` and `slice` methods were inspired by Python's sequence operations. The Array `push`, `pop`, `shift`, `unshift`, and `splice` were directly modeled on like-named Perl array functions. Python also inspired the String `concat`, `slice`, and `search` methods while String `match`, `replace`, and `substr` came from Perl. Java inspired `charCodeAt`. The syntax and semantics of regular expression string matching was borrowed from Perl.

The statement-level additions provide previously missing statements that programmers familiar with C-family languages would expect. The `do` statement directly replicates the syntax and analogous semantics of the C `do` statement that was left out of JavaScript 1.0. Labeled statements and `break` or `continue` naming a label is directly modeled after the same feature in Java. They enable multilevel early escapes from nested iteration and `switch` statements and early escapes from non-iterative code blocks. JavaScript 1.2's `switch` statement includes compile-time evaluation of `case` selector expressions [Eich et al. 1998, jsemit.c lines 757–776] as in C and Java.

In JavaScript 1.0/1.1 functions could be defined only by global declarations at the top level of scripts. JavaScript 1.2 permits functions to be defined using local declarations within another enclosing function. Such inner function definitions can be nested to an arbitrary level. Inner functions are lexically scoped and their local declarations shadow identically named declarations in outer scopes. In JavaScript 1.0/1.1 forward referencing of variables and functions was possible because the language logically "hoisted" top-level `var` and `function` declarations to the beginning of their script and function local `var` declarations to the beginning of the function body. In JavaScript 1.2 nested `function` declarations are also hoisted to the beginning of the enclosing function body. If there is more than one `function` declaration with the same name, the one that occurs last in the source code of the enclosing function body is bound to the name.

JavaScript 1.2 also provides lambda expressions by allowing function definitions to occur as expression primaries. They are called "function expressions" and are syntactically identical to `function` declarations except that the function name is optional. If the name is present, the function expression is treated as a hoisted `function` declaration for binding purposes. A function expression without a function name defines an anonymous function. In either case, each runtime evaluation of the function expression creates a new closure. The addition of the `callee` property to the `arguments` object permits such closures to recursively reference themselves.

Array literals and object literals[28] were inspired by similar features in the Python language. Array literals provide a concise syntax for creating and initializing the elements of an Array object. Array literals enable a JavaScript programmer to write the following:

```
var p2 = [1,2,4,8,16,32,64];
```

JavaScript 1.2

instead of the following:

```
var p2 = new Array();
p2[0] = 1;
p2[1] = 2;
p2[2] = 4;
// etc.
```

JavaScript 1.1

Similarly, object literals provide a concise syntax for creating an object and associating properties with it. Using an object literal, a programmer can write the following:

```
var origin = {x: 0, y: 0};
```

JavaScript 1.2

instead of the following:

```
var origin = new Object;
origin.x = 0;
origin.y = 0;
```

JavaScript 1.0

The combination of object literals and function expressions make it easy to define classless objects that include methods, such as the following:

```
function Point(x, y) {
  return {
    x: x,
    y: y,
    distance: function (another) {
      return Math.sqrt(Math.pow(this.x - another.x, 2)
                       +Math.pow(this.y - another.y, 2));
    }
  }
  var origin = new Point(0, 0);
  alert(origin.distance(new Point(5, 5));
```

JavaScript 1.2

[28]The JavaScript 1.2 documentation and the ES3 specification called these "array initializers" and "object initializers." But the "literal" terminology is more common among JavaScript programmers and in articles and books.

Combining object literals and function expressions also provides a more convenient way to define prototype objects. Also added is the `__proto__` pseudo-property that enables a JavaScript program to dynamically access and modify the internal reference each object uses to access inherited properties.[29] Using `__proto__` a program can dynamically construct arbitrarily deep property inheritance hierarchies and dynamically change whence an object obtains inherited properties.

Some JavaScript 1.2 changes ultimately proved to be missteps. The `import` and `export` statements were intended for use with a Java-compatible script signing mechanism [Netscape 1997a] provided in Netscape 4. Globals defined in a signed script were private to that script except for functions that were explicitly exported using the `export` statement. This features was never adopted by non-Netscape browsers.

Even though user requests had motivated the JavaScript 1.0/1.1 == operator's coercion rules, some users were finding that behavior surprising and confusing. Brendan Eich decided to fix == in JavaScript 1.2 by eliminating most of its automatic coercions [Netscape 1997d; Rein 1997]. If both operands are not of the same primitive type (number, string, Boolean, object) == would return `false`.

The hope with JavaScript 1.2 was that use of the `<script> version` attribute would be sufficient to deal with the changes to JavaScript 1.0 and 1.1 semantics. But by the time of the JavaScript 1.2 production release this form of versioning was already becoming difficult for Web developers to manage [Rein 1997]—particularly for Web pages that needed to also work with non-Netscape browsers with their own implementations of JavaScript.

6 INTERLUDE: CRITICS

From its earliest days, JavaScript has been the target of intense criticism. Some of the criticism has been directed at fundamental design decisions such as its use of dynamic typing or design details such as its coercion rules. Other critics have fundamental disagreement with how it integrated with HTML or concerns about its exposure of browser security vulnerabilities [Fair 1998]. Robert Cailliau [Wikinews 2007] called JavaScript "the most horrible kluge in the history of computing" and said, "I know only one programming language worse than C and that is Javascript [*sic*]." Bret Bos [2005], at a W3C workshop, characterized JavaScript as "the worst invention ever."

For many novice programmers, JavaScript in browsers is their first exposure to common programming issues, such as the challenges of floating point arithmetic. They typically assume that those problems are unique to JavaScript. Many experienced programmers compare JavaScript to familiar programming languages (or to Java, because of the name confusion) and find it lacking. Articles [Cardy 2011] that catalog JavaScript's quirks and websites such as wtfjs.com [Leroux 2010] became a Web staple.

Part 2: Creating a Standard

7 FINDING A VENUE

When the Mocha project began in 1995 it was already clear that standards would be needed to ensure the interoperability of Web pages across different Web browsers. This was formally recognized in the Netscape and Sun [1995] JavaScript announcement:

> Netscape and Sun plan to propose JavaScript to the W3 Consortium (W3C) and the Internet Engineering Task Force (IETF) as an open Internet scripting language standard.

However, neither the W3C nor the IETF were a suitable venue for creating a vendor-independent JavaScript specification. The IETF focus was on Internet protocols and data formats, rather than

[29] The `__proto__` pseudo-property is similar to a Self parent slot.

programming languages. The W3C was a new organization, and its technical leadership was not interested in adding an imperative programming language to the Web technology suite. As Berners-Lee's collaborator Robert Cailliau recounted in an interview [Wikinews 2007]:

> For example, I was convinced that we needed to build-in [*sic*] a programming language, but the developers, Tim [Berners-Lee] first, were very much opposed. It had to remain completely declarative.

In early 1996 the evolution of browser technologies was racing on "Internet time"[30] [Iansiti and MacCormack 1997] yet language standardization had a reputation for being a slow and often contentious process. With Microsoft taking browser competition seriously, Netscape and Sun feared that Microsoft might try to dominate development of Web scripting standards and attempt to refocus it on a Visual Basic-based language. In the spring of 1996, Netscape and Sun needed to find a recognized standards development organization under whose umbrella a JavaScript standard could be quickly drafted with Microsoft participation but not Microsoft domination. Carl Cargill, a standards expert working for Netscape, knew the Secretary-General of Ecma International, Jan van den Beld, and steered JavaScript standardization toward it. Ecma positions itself as a business-focused standards organization that minimizes bureaucratic processes in order to minimize standards development time. The International Standards Organization recognizes Ecma International, and Ecma standards can use a fast-track process to become ISO standards. In addition to Cargill's connections, Sun was already an Ecma member and considered Ecma to have proven its independence by publishing a Windows API standard over Microsoft's objection [LaMonica 1995].

Informal contacts and discussions involving Netscape, Sun, and Jan van den Beld, took place over the spring and summer of 1996. In September, the Ecma Co-ordinating[31] Committee [1996b] considered a Netscape request to start a JavaScript standardization activity and authorized a start-up meeting targeted for November 4–5, 1996, in Silicon Valley. Netscape formally applied [Sampath 1996] for Ecma membership as an Associate Member.[32] On October 30, an open invitation [Ecma International 1996a] for a "start-up meeting on a project on JavaScript" was published. A new Ecma Technical Committee would be organized for the activity if there was sufficient interest. Ecma uses numeric designators for its technical committees and the next available number was 39. In December 1996, at its semi-annual meeting the Ecma General Assembly approved the creation of TC39 and a Statement of Work. At the same time, Microsoft joined Ecma as an Ordinary Member.

8 THE FIRST TC39 MEETING

The TC39 organizing meeting took place November 21–22, 1996, at the Netscape offices in Mountain View, California. The minutes [TC39 1996] record that there were thirty attendees (Figure 11). The meeting opened with welcomes from Jan van den Beld on behalf of Ecma and David Stryker, the Netscape VP of Core Technologies. Stryker expressed his wish that the committee would create a specification with minimal deviations from the then-current implementations and that language extensions beyond that should be deferred to the future.

Thomas Reardon, the leader of the Microsoft Internet Explorer development team, recommended that the committee "avoid duplication" by not working on the built-in library for an HTML object model and instead leave that to the W3C. This recommendation was accepted by the committee and was essential to the early success of the committee as Netscape's and Microsoft's core language

[30] A term coined to describe the short development cycles and frequent product release of Netscape and other early Web technology developers.

[31] During this period, Ecma spelled "coördinating" using an explicit hyphen.

[32] Ecma Associate Members participate in one Technical Committee. Ecma calls its highest membership level "Ordinary Members." Ordinary Members are full voting members of the Ecma General Assembly and may participate in all TCs.

Acting Chairman:	Mr. J. van den Beld
Secretary:	Mr. J. van den Beld (SG ECMA)
Attending:	Mr. Cargill (Netscape), Ms. Converse (Netscape), Mr. Eich (Netscape), Mr. Fisher (NIST), Mr. Gardner (Borland), Mr. Krull (Borland), Mr. Ksar (HP), Mr. Lenkov (HP), Mr. Lie (W3C), Mr. Luu (Mainsoft), Mr. Mathis (Pithecanthropus, JTC1/SC22), Mr. Matzke (Apple), Mr. Murarka (Spyglass), Ms. Nguyen (Netscape), Mr. Noorda (Nombas), Mr. Palay (Silicon Graphics), Mr. Reardon (Microsoft), Mr. Robinson (Sun), Mr. Singer (IBM), Mr. Smilonich (Unysis), Mr. Smith (Digital), Mr. Stryker (Netscape), Ms. Thompson (Unisys), Mr. Urquhart (Sun), Mr. Veale (Borland), Mr. Welland (Microsoft), Mr. White (AAC Group, Microsoft), Mr. Willingmyre (GTW Associates, Microsoft), Mr. Wiltamuth (Microsoft).
Excused:	Mr. Huffadine (Callscan)

Fig. 11. Attendees at the first meeting of TC39—as recorded in the meeting minutes [TC39 1996]

features were very similar but their HTML APIs were very different. The decision that TC39 would develop only platform/host-environment independent standards became and continues to be one of TC39's core operating principles. Reardon discussed the difficulties Microsoft had experienced in trying to make JScript fully compatible with Netscape and stressed the need for formalization of a language specification. But he also cautioned that the specification should leave room for competing implementations to add value.

The proposed agenda had included technical presentations from Netscape, Sun, Microsoft, and Nombas Inc. as well as the actual organizational activities needed to set up a new Ecma technical committee and start work on drafting a standard language specification. But at the meeting Sun said that it did not need to present anything and a Borland International presentation was added to the agenda.

Both Netscape and Borland had handed out draft technical specifications at the beginning of the meeting. Microsoft did not. During Thomas Reardon's presentation he said that Microsoft had developed its own preliminary specification and held up a document. Reardon claimed that they had not had time to get it copied yet but would have copies available the next day so the Microsoft technical presentation was moved to the second day of the meeting.

Brendan Eich attended, but the Netscape technical presentation was made by Anh Nguyen. It introduced the preliminary draft of *JavaScript Language Specification* for JavaScript 1.1 authored by Eich and C. Rand McKinny [1996]. Netscape contributed this document to Ecma to serve as one of the base documents for the standardization effort. Nguyen explained that JavaScript 1.1 in Netscape Navigator 3 had a few deviations from the initial JavaScript version in Netscape 2. The Netscape specification described the language syntax using a BNF notation similar to that used in the ANSI C language standard [ANSI X3 1989]. It used informal prose to define most semantics and a table formulation to describe the language's coercion rules.

Borland had created a server implementation of JavaScript and a JavaScript IDE [Lazar 1997]. Their presentation focused on several language extensions they had already made or intended to make in their implementation [Borland International 1996]. The major extensions were class definitions, try/catch/finally exception handling, a C-like switch statement, code-blocks as first-class values, array literals, a C-like preprocessor, and a number of additions to the built-in library including some I/O functionality. Borland also noted the difficulties they had in trying to achieve compatibility with Netscape's implementation and said that a more formal specification was needed in order to ensure interoperable implementations.

Brent Noorda of Nombas Inc. presented Nombas' experience with its Cmm ("C minus minus") product that was marketed as a scripting language. The surface syntax and some of the semantics of Cmm were quite similar to JavaScript 1.0 and Nombas subsequently evolved its Cmm implementation into an ECMAScript implementation for embedded applications [Noorda 2012].

As soon as the first day of the meeting adjourned, Microsoft's Robert Welland had work to do [Welland et al. 2018, at +8:30]. Thomas Reardon's claim of not having time to make copies was a stalling tactic to give Welland more time to work on Microsoft's specification. The task of creating a specification document for the meeting had been assigned to a Microsoft technical writer and when Welland received it, just before traveling to attend the meeting, he found that the document was inadequate as even a preliminary language specification. He did not want to give it to the committee. But when he saw the Netscape document before the start of the meeting he felt is was also inadequate and did not want it to become the sole base document for developing a standard. Welland and Reardon decided they would stall for a day with the intent of having a better document ready at the start of the second day of the meeting.

After the meeting, Robert Welland went to the home of his former NewtonScript colleague, Walter Smith, who was also working for Microsoft but was still located in the Bay Area. They worked through the night turning the Microsoft document into a plausible preliminary specification of the core JavaScript language. Their specification also borrowed much of the grammar from the ANSI C standard and used a table for coercion rules. But Welland wanted the rest of the semantics to be more formally specified. He recalled [Welland et al. 2018, at +10:10] that the *LISP 1.5 Programmer's Manual* [McCarthy and Levin 1965] had described the semantics of the Lisp interpreter using a style where a syntactic form was immediately followed by a precise description of how that syntax should be evaluated. In some cases, the semantics were presented using pseudocode.[33] Welland decided to use a similar style of pseudocode with numbered steps to describe the evaluation semantics of JavaScript.

Welland and Smith added semantics to the document based upon the then-current JScript implementation. Where there were things they were uncertain about they fell back to their Self and NewtonScript experience and described what made sense from that perspective. The document includes an object diagram of an Array that looks very Self-like in how it models property inheritance. By the next morning they had a document that they felt was good enough to hand out as a starting point. They made copies and Welland distributed them at the start of the second day of the meeting. That document was *The JScript Language Specification, Version 0.1* [Welland et al. 1996] and became Microsoft's base document contribution to the Ecma effort.

When Robert Welland made his presentation he was pleasantly surprised that the meeting attendees generally liked his document and agreed that a more formal specification was needed in order to ensure interoperable implementations. However, the consensus was to not wait for another as yet unidentified formal specification technique but instead to create the initial draft of a standard by integrating the Netscape, Microsoft, and Borland specification contributions and then working to make the resulting draft more complete and precise. As a first step the committee created an issues list [Appendix G] of items that needed to be resolved or clarified for the first version of the standard. Because there were two proposed base document submissions, one of them had to be selected as the document to start editing upon. Netscape's document was authored using FrameMaker and the Microsoft document was authored using Word. The Ecma representatives explained that their internal editorial processes use Word so, to Welland's surprise, the committee agreed that they would use the Microsoft contribution as the base document.

[33] Welland may have been thinking of McCarthy's Appendix B's description of the PROG feature.

Chairman:	**Mr. G. Robinson** (Sun)
Vice-Chairman:	**Mr. C. Cargill** (Netscape)
Vice-Chairman:	**Mr. S. Wiltamuth** (Microsoft)
Principal editor:	**Mr. M. Gardner** (Borland) (to be confirmed)
Assistant editor:	**Mr. A. Murarka** (Spyglass) (to be confirmed)

Fig. 12. Officers Elected at First TC39 Meeting [TC39 1996]

The committee selected an initial slate of officers (Figure 12) and set very aggressive goals of having a first draft ready for the next meeting in January 1997, a final draft in April 1997, and targeting approval of the standard by the Ecma General Assembly in June 1997. They scheduled subsequent meetings at approximately six-week intervals and initiated actions to set up a private mailing list and ftp server.[34]

The second meeting of TC39 [1997e] was on January 14–15, 1997. There were twenty-two attendees including five guests who were not affiliated with Ecma members. Jan van den Beld announced that the establishment of TC39 had been confirmed by the Ecma General Assembly. He stressed that as soon as possible TC39 needed to start following the Ecma rules concerning membership and participation. Contributors to the development of Ecma standards must be representatives of organizations which are Ecma members.

The main technical content of the meeting was a review and discussion of the first draft [TC39 1997c] of the standard. Borland's Michael Gardner and Randy Solton had created the document by merging and integrating the contributions from Netscape, Microsoft, and Borland. Spyglass did not join Ecma so Anup Murarka did not participate in developing the first draft. Features that were exactly the same in all three implementations were considered uncontroversial, and places where features differed were identified for reconciliation.

Features that were unique to a given implementation were listed in a "Proposed Extensions" appendix. The committee discussed how to handle extensions and agreed to prioritize the core features common to all then-current implementations over any extensions. There was also agreement that the specification should avoid changes that would require modifying existing applications. This ultimately became an important design guideline for all future revisions of the standard.

In order to meet the tight schedule, the committee established an ad hoc technical working group that was authorized to work with the editor to fill in missing material and resolve outstanding technical issues within the specification. The group was to communicate electronically and to meet weekly alternating between face-to-face meetings and teleconferences. Scott Wiltamuth was to serve as rapporteur. The TC39 meeting adjourned at 10:30 AM on January 15 and the rest of the day was used for a meeting of the ad hoc technical working group.

After the meeting, Borland decided to not join Ecma so Michael Gardner could not continue as the editor. Sun made Guy Steele available and he served as editor starting in late January 1997 and continuing through publication of the standard's first edition in September 1997.

9 CRAFTING THE SPECIFICATION

Michael Gardner and Randy Solton started work on the first draft specification immediately after the November meeting and remarkable progress was made over the next six weeks. In addition to Gardner and Solton, the first draft lists the following technical contributors: Brendan Eich (Netscape), C. Rand McKinney (Netscape), Donna Converse (Netscape), Shon Katzenberger (Microsoft), and Robert Welland (Microsoft).

[34] We have not located any archives of these channels.

January 10, 1997 Draft	**ECMA-262, First Edition**
	Scope
	Conformance
	Reference
	Overview
Notational Conventions	Notational Conventions
Source Text	Source Text
Lexical Conventions	Lexical Conventions
Types	Types
Type Conversion	Type Conversion
Variables	Execution Contexts
Expressions	Expressions
Statements	Statements
Function Definition	Function Definition
Program	Program
Native ECMAScript Objects	Native ECMAScript Objects
	Errors

Fig. 13. Organization of the ECMAScript Specification

Robert Welland had returned to Redmond and handed off his JScript 0.1 specification to Shon Katzenberger to continue developing the language semantics [Welland et al. 2018, at +12:02]. Katzenberger, a Mathematics PhD, was comfortable with formal notations and found the pseudocode concept (Appendix Q) worked well for describing the JavaScript semantics. It provided a level of detail that he thought would be sufficient for ensuring interoperability. Katzenberger became Microsoft's primary technical contributor to the development of the standard. He expanded upon Welland's and Smith's late-night work by checking it against the running implementations and writing additional pseudocode algorithms for parts of the language they had not covered. He then passed his new and revised material on to the Borland editors to incorporate into the official draft. Katzenberger, when interviewed in 2018 [Welland et al. 2018, at +21:16], recalled being unhappy that editorial changes sometimes unintentionally broke his algorithms. He was very pleased when Guy Steele became available to serve as editor.

The January 10 draft [TC39 1997c] established the basic structure of the specification (Figure 13) and defined many of the basic techniques, conventions, and verbiage used to define the language. Many of these were still in use twenty years later in editions of the ECMAScript standard.

The description of the grammar conventions was largely taken from the Netscape specification but the structure of the expression and statement level grammars and the production names largely follows what was used in Microsoft's specification. The expression grammar is different from both contributed specifications in subtle details such as the precedence of function call, object creation (`new` operator), and object property access expression elements.

The draft attempts to precisely specify the rules of automatic semicolon (ASI) insertion as a parsing error-correction procedure. The statement grammar includes explicit semicolons that terminate all non-compound statements. Without ASI, missing semicolons would produce parse errors. The ASI specification defines when a JavaScript parser must try to correct such parse errors by assuming the presence of a semicolon and reparsing. The ASI rules in the first draft were incomplete and were refined in subsequent drafts and editions of the ECMAScript specification.

The January 10 draft includes Shon Katzenberger's pseudocode algorithms, such as the one in Figure 14, to define the semantics of various language constructs. The algorithms consisted of sequential numbered steps and simple conditional control flows among the steps. Each step consists

4.7.4 GetValue(*V*)

1. If Type(*V*) is not a Reference, return *V*.
2. Call GetBase(*V*)
3. If Result(2) is **null**, generate a runtime error.
4. Call the [[Get]] method of Result(2), passing GetProperty(*V*) for the property name and GetAccess(*V*) for the access mode.
5. Return Result(4).

Fig. 14. A Named Pseudocode Algorithm from the Jamuary 10, 2007, draft ECMAScript specification [TC39 1997c, § 4.7.4]. A period at the end of step 2 is missing in the original document.

of an *imperative*[g] prose statement. The prose of a step is written in English incorporating a basic vocabulary defined by the specification for commonly occurring actions. Algorithms may be named and "called" from other algorithms within the specification.

This draft also defines the data types used within the algorithms: Number, Boolean, String, Object, Undefined, and Null are the types of values visible to an ECMAScript program. Reference, Completion, and List type values are used to define language semantics and are not directly observable to ECMAScript programs.

The specification of the Object type introduces the concept of a property *attribute*[g] that control how individual properties may be accessed or modified. It defines seven different attributes: ReadOnly ErrorOnWrite, DontEnum, NotImplicit, NotExplicit, Permanent, and Internal. Ultimately ErrorOnWrite, NotImplicit, and NotExplicit were eliminated and Permanent was renamed as DontDelete. Properties with the Internal attribute hold internal state that is associated with objects but is not directly visible to an ECMAScript program. Such *internal properties*[g] are used to hold state needed to implement the object semantics or unique behaviors of built-in and host objects.

Also introduced is the concept of *internal methods*[g], which are algorithms that define the essential behaviors of objects. Different kinds of objects (for example, Array objects) may be specified using alternative definitions for some internal methods thus providing them with behavioral variations. The internal method interfaces are essentially the specification of a simple *metaobject protocol*[g] [Kiczales et al. 1991].

Within the specification, internal methods and internal properties have names enclosed in double brackets such as [[Foo]]. The January 10 draft defined the internal methods [[Get]], [[Put]], [[HasProperty]], [[Construct]], [[Call]], and the internal property [[Prototype]]. Those internal methods were used in the first formalization of the semantics of object property access, prototype inheritance, and function invocation. By the time ES1 was completed [[CanPut]] and [[Delete]] internal methods were added.

The first draft's table of contents includes sections for native (built-in) ECMAScript objects and for objects provided by browser and Web server hosting environments. But these sections were empty in the January 10 draft. There were twenty items explicitly tagged "Issue" within the draft. These are in addition to a number of possible language extensions which are described in an appendix.

The January 10 draft was the basis for discussion at the first technical working group meeting on January 15, 1997. Several important decisions were made [Wiltamuth 1997a], including the following:

- The initial standard would not include specifications for host-specific library objects and functions such as provided by browser and Web server hosts.

Scott Wiltamuth (note taker)	Microsoft
Brendan Eich	Netscape
Shon Katzenberger	Microsoft
Michael Gardner (1st draft co-editor)	Borland
Randy Solton (1st draft co-editor)	Borland
Clayton Lewis	Netscape
Guy Steele (editor)	Sun

Fig. 15. ES1 Specification Working Group Regular Participants

- Extensions to the then-current language would be considered only after a complete draft specification was available.
- The comma and ? operators do not propagate Reference values hence they cannot be used on the left hand side of assignment and do not provide a `this` value to function calls.
- Non-ASCII Unicode characters would not be allowed in identifiers.
- NUL (U+0000) characters are allowed in string values.
- Global function and variable declarations create properties which are enumerable and deletable. Specification defined properties of built-in objects default to non-enumerable and deletable.

Open issues that were not resolved at that first working group meeting included order of evaluation of multiple assignments, the semantics of assignment to an inherited read-only property, and how to accommodate date values prior to 1970.

The working group (Figure 15) met regularly through mid-April 1997. The group worked through a list of major and minor issues and reviewed working draft text as it was prepared by the editor. Notes exist for nine working meetings [Wiltamuth 1997a,b,c,d,e,f,g,h,i]. Richard Gabriel, who attended some of the working group meetings, recalled in a personal communication a not uncommon interaction during these meetings. Guy Steele would ask a question about some edge-case feature behavior. Sometimes Brendan Eich would say "I don't know," and sometimes Eich and Shon Katzenberger would be unsure or disagreed; in such cases they would each turn to their respective implementation and try a test case. If they got the same answer, that became the specified behavior. If there was a difference they would discuss the issue until they reached an agreement.

Following the first Gardner/Solton specification draft, seven additional drafts were released to the full committee by Guy Steele between February 27 and May 2, 1997, with additional working drafts circulated within the working group. Except for the final draft [TC39 1997b] for the Ecma General Assembly, each draft contained a detailed issue resolution log [TC39 1997d].

Some issues had a long-term impact upon the language and its usage. For example, one issue of continuing discussion was whether the short-circuiting Boolean operators `&&` and `||`, when presented with Boolean-coercible operands, evaluated to the actual value of one of the operands ("Perl-style") or a Boolean `true` or `false` value ("Java-style"). Brendan Eich had originally implemented them mostly with the "Perl-style" semantics but with "Java-style" behavior in a few cases. Microsoft and Borland had implemented the full "Java-style" semantics. The ultimate decision was to consistently apply the "Perl-style."

That decision directly enabled what a few years later became a widely used JavaScript idiom. Values `null` and `undefined` are coerced to `false` by Boolean operators and all Object references coerce to `true`. This led to the idiom shown in Figure 16 for providing a default value for object properties and optional function parameter.

```
function f(options) {
  options = options || getDefaultOptionsObject();
  // if an options object was passed, use it.
  // otherwise use the default set of options
  ...
}
```

ES1

Fig. 16. ECMAScript 1 enabled idiom for providing a default value for a function parameter.

Brendan Eich recalls that he hoped to include his JavaScript 1.2 changes to the == operator semantics that eliminated its type coercions. Shon Katzenberger successfully argued that it was too late to make such a change given the large number of existing Web pages it would break. Eich reverted to the original semantics in the JavaScript 1.3 release of SpiderMonkey.

The third meeting of TC39 was March 18–19, 1997. This was the last scheduled formal TC39 meeting prior to the June Ecma General Assembly where, it was hoped, the first edition of the standard would be accepted and approved. In order to meet that schedule TC39 would need to vote at its third meeting to refer the standard to the GA.

On the March 12, draft version 0.12 [TC39 1997a] was distributed to the full committee and discussed [TC39 1997f] at a working group meeting on March 14. This draft was nearing technical completion except that the complex definition of Date object was still simply a set of section headings. Shon Katzenberger had a complete specification-quality proposal which after discussion and review could be incorporated into the specification. The document had grown from forty-one substantive pages to ninety-six pages in the two months since completion of the January 10 draft. Draft 0.12 had eight internal "Issue" tags and six significant entries in its issue-tracking appendix in addition to the missing Date specification. About a dozen other issues were discussed at the working group meeting and would need to be addressed in the specification.

Based upon Scott Wiltamuth's assurance that no contentious issues remained and that a complete draft could be finished by the end of March, TC39 unanimously agreed to forward a draft to the Ecma General Assembly for a June approval vote. The Specification working group was given the responsibility to finish the specification and work with the Ecma Secretariat staff to produce a final draft that met their schedule and formatting requirements. Completing the draft took a month longer than Wiltamuth projected. Three more intermediate drafts were internally distributed before the final draft [TC39 1997b] was finished on May 2, 1997. It was distributed to the General Assembly members on May 5. The final draft conformed to Ecma's document conventions and included a *non-normative*[g] overview of the language written by Richard Gabriel. At its June 1997 meeting, the General Assembly agreed to a publish the draft as *Ecma Standard ECMA-262, 1st Edition* after some minor editorial changes, and to submit it into the ISO fast track process. The editorial changes were completed and distributed to TC39 on September 10, 1997. *ECMA-262, 1st Edition* was released for publication [Steele 1997] at the September 16–17 TC39 [1997h] meeting.

10 NAMING THE STANDARD

From the start of the standardization process, it was understood that the name of the language was going to be problematic. Netscape's initial name, "LiveScript," had been replaced by "JavaScript" as part of its strategic partnership with Sun. Sun trademarked "JavaScript" and licensed the trademark to Netscape. While Sun was supportive of the standardization effort for Netscape's scripting

language they were also aggressively protecting their Java-related trademarks. It seemed unlikely that Sun would relinquish control of the "JavaScript" trademark to a standards organization.

At the first TC39 meeting the attendees invited Sun to contribute the name "JavaScript" and agreed to use "ECMAScript" as a placeholder name until a more suitable name could be found. Scott Wiltamuth was assigned the task of collecting name suggestions and checking their availability.

Wiltamuth [1997j] presented a list of sixteen potentially viable names and fourteen names that were thought to be nonviable because of existing trademarks or conflicting usages. A straw poll identified the top candidate names: LiveScript, ScriptJ, EZScript, Xpresso/Expresso/Espresso. The Netscape and Sun delegates were asked to investigate the availability of LiveScript and JavaScript. In the interim, the "ECMAScript" continued to be used in the specification drafts.

Sun confirmed [TC39 1997f] that it would not license "JavaScript" to Ecma. Netscape stated it had no legal objection[35] to the use of the name LiveScript for the standard. Based upon that feedback, TC39 agreed to work with Netscape to secure the rights to LiveScript and that Ecma would investigate trademark registration. However, ECMAScript would still be used in the specification drafts until written confirmation was received from Netscape.

The draft standard that was submitted to the Ecma General Assembly still used ECMAScript as the name of the language. At the GA meeting [Ecma International 1997] there were concerns about the appropriateness of using a trademarked name in the title of a standard as the intent of a standard is to place all companies implementing the standard on equal footing. This precluded the use of LiveScript as the name of the language as Netscape had decided that it was not willing to formally transfer the name to Ecma. The general assembly approved the standard with the placeholder "ECMAScript" name and instructed TC39 to resolve the naming issue by September.

Naming was discussed at the July TC39 [1997g] meeting. Scott Wiltamuth proposed "RDScript"[36] and Carl Cargill proposed adopting "ECMAScript" as the permanent name. There was discussion whether any name was actually necessary. Perhaps "ECMA-262," the Ecma document number for the specification would suffice as a name. Ultimately, nothing was resolved at the July meeting but in September TC39 [1997h] agreed on publishing the standard using "ECMAScript" as the language name.

A few months later ANSI, the American national standards body, in casting its vote on approval of ECMA-262 as an ISO standard made this comment [TC39 1998e]: "it is unlikely that any implementation of the language will be called ECMAScript. This has and will result in confusion for users as to what the standard means and what Language engines support the standard." This prediction has proven to be largely correct. The world at large has continued to use the name "JavaScript" to identify the language that is implemented by browsers and that name is enshrined in the specification of the HTML `<script>` element. Brendan Eich [2006b] later expressed his opinion of the naming issue: "ECMAScript was always an unwanted trade name that sounds like a skin disease."

11 ISO FAST-TRACK

The final step in the initial standardization of JavaScript was to get the Ecma specification accepted as an International Standards Organization (ISO) standard. In September 1997, the first edition of ECMA-262 was submitted into the ISO/IEC fast-track process [TC39 1997h]. Guy Steele subsequently resigned as Project Editor and was replaced by Mike Cowlishaw of IBM.

[35] Brendan Eich believes that Netscape was never serious about allowing Ecma to use the name LiveScript.
[36] Rapid Development Scripting language

An ISO/IEC ballot produced twenty-seven pages of comments [TC39 1998e] from the national standards bodies of Denmark, France, Japan, Netherlands, and the USA. Also included were comments that TC39 [1998b] submitted listing errors it had found. The majority of the comments identified minor editorial issues that had been missed during the rapid creation of ECMA-262. There were also a few, more significant, technical issues reported relating to the Date object's year 2000 transition support and with the integration of Unicode into the language.

Mike Cowlishaw, with TC39 input, prepared a Disposition of Comments Report that was reviewed and accepted at a ballot resolution meeting [TC39 1998a]. In July 1998, the camera-ready revised specification was released to ISO/IEC and a postal ballot was sent to Ecma ordinary members who approved the revised specification as *ECMA-262, 2nd Edition* [Cowlishaw 1998].

12 DEFINING ECMASCRIPT 3

At the first TC39 meeting a number of extensions to the JavaScript 1.0/1.1 language were proposed and some were incorporated into the first draft of the language specification. But the TC39 technical working group had agreed to defer consideration of any new features until specification of the base language was complete. For most of the development of the first edition, possible extensions were relegated to an appendix of the draft specification [TC39 1997a, Appendix B].

By the July 1997 TC39 [1997g] meeting, work on the first edition was nearly complete. The committee's focus shifted to considering what new features should be incorporated into the next edition of the specification. Netscape had already indicated its direction with the shipment of Netscape 4.0, incorporating the SpiderMonkey engine with the JavaScript 1.2 extensions. Scott Wiltamuth presented Mircosoft's [1997] initial proposal for "ECMAScript 2.0" that included a `switch` statement, a `do while` statement, and labeled statements with labeled `break` and `continue`. Also included were the `===` and `!==` operators and adding the `caller` property to the `arguments` object. Andrew Clinick [1997] of Microsoft presented a separate proposal for adding conditional compilation support. The starting point for "Version 2" solidified in October when Microsoft shipped JScript 3.0 as a component of Internet Explorer 4.0. Figure 17 lists the major extensions to ECMAScript 1st Edition implemented by the Netscape [1997c] and Microsoft [2009b] browsers as of the end of 1997.

The formal TC39 meetings had become management and strategy sessions attended by group and program managers representing the member companies. Most of the technical work of the committee occurred in informal technical working groups. At the July meeting TC39 agreed on a set of steps for developing Version 2. It also agreed that it was the responsibility of the technical working groups to define the work items, feature proposals, and acceptance criteria. Version 2 would be allotted more time than had been available for the first edition to allow the draft to mature and receive external feedback. The target date for a first draft of the Version 2 specification was December 1997. At the September meeting [TC39 1997h] it was further agreed the Version 2 specification must be backward compatible with programs that comply with the 1st edition specification.

At the times these decisions were made, the ISO fast-track process had not yet started and it was not anticipated that the resulting changes would require a new edition of ECMA-262 to align with the ISO edition. By early 1998, there were two working groups with overlapping membership working on two separate specification drafts. It became apparent that "Edition 2" and "Version 2" were not going to be the same publication. However, TC39 delegates continued to call the next round of feature work "Version 2" or "V2" even after it was known that it would likely be published as the "3rd Edition." This was not the last time where TC39's internal version naming would end up clashing with its ultimate publication nomenclature.

At the end of 1997, there were major participation changes in the technical working groups. Figure 18 lists individuals appearing in the meeting notes for at least two working group meetings

Feature	JavaScript 1.2	JScript 3.0	ECMA-262 3rd Edition
do statement	✓	✓	✓
break/continue to label	✓	✓	✓
switch statement	✓	✓	✓
Nested functions	✓	✓	✓
Functions in expressions	✓	✓	✓
Object literals	✓	✓	✓
Array literals	✓	✓	✓
=== and !==		✓	✓
Regular Expression literals	✓	✓	✓
delete operator	✓	✓	✓
__proto__ pseudo property of all objects	✓		
Array methods: concat, slice	✓	✓	✓
Array methods: push, pop, shift, splice, unshift	✓		✓
Sparse arrays with inherited elements	✓		✓
String methods: fromCharCode, match, replace, search, substr, split using regular expressions	✓	✓	✓
String method: charCodeAt	✓		✓
RegExp methods: compile, exec , test	✓	✓	✓
RegExp properties: $1...$9, input	✓	✓	
RegExp global properties: lastMatch, lastParen, leftContext, rightContext	✓ ✓		
arguments object has local declaration properties	✓	✓	
arguments.callee	✓		✓
arguments.caller	✓	✓	
watch/unwatch functions	✓		
import/export statements and signed scripts	✓		
Conditional compilation		✓	
debugger keyword		✓	

Fig. 17. Extensions to *ECMA-262 1st Edition* that shipped in major Web browsers in 1997. Most of these were ultimately included in *ECMA-262 3rd Edition*.

over the course of 1998. Of the regular participants in the working group that developed the first edition, only Clayton Lewis remained active. Brendan Eich attended one meeting in February 1998 and then became a co-founder of the Mozilla Project [Mozilla Organization 1998], the effort to open source the code of the Netscape browser. Waldemar Horwat assumed the role as Netscape's language design lead for TC39. Similarly, Microsoft's Katzenberger took a sabbatical and then moved on to other projects. Herman Venter and Rok Yu assumed his TC39 responsibilities for Microsoft.

In October 1997, the technical working group produced lists (Appendix H) of features that were candidates for inclusion in Version 2. The features listed as having agreement for inclusion were largely the union of the Netscape JavaScript 1.2 and Microsoft JScript 3.0 features with a few exclusions. It also included toSource. This corresponded to an object serialization/persistence scheme that Brendan Eich had developed for JavaScript 1.3.[37] Other speculative features that lacked

[37]The serialization scheme included an extensible set of toSource methods for serializing individual objects as JavaScript source code and "sharp variables" for representing circular references. The global function uneval would serialize an object graph starting with a root object. The resulting source code string could be deserialized using eval. Brendan Eich borrowed the sharp variable syntax #n= and #n# from Common Lisp[Steele 1990, pages 578–579].

Norris Boyd	Netscape	Drew Lytle	Microsoft
Andrew Clinick	Microsoft	Karl Matzke	SunSoft
Mike Cowlishaw	IBM	Mike McCabe	Netscape
Jeff Dyer	Nombas	Dave Ragget	HP/W3C
Bill Gibbons	Netscape	Herman Venter	Microsoft
Waldemar Horwat	Netscape	Rok Yu	Microsoft
Mike Ksar	HP	Chris Weight	Microsoft
Clayton Lewis	Netscape		

Fig. 18. Regular Participants, 1998 TC39 Technical Working Group

consensus for inclusion were listed separately. As with Edition 1, much of the working group's attention would be focused on precisely specifying already implemented features and resolving any differences that existed between the implementations. But the agreed feature list also included an exception handling mechanism, an `instanceof` operator, and other features that were not yet part of any implementation. These would require a form of design work that had not been necessary for the first edition. Figure 19 lists features that were not in pre-1998 browsers which were ultimately included in ES3.

The technical working group established a rhythm of monthly face-to-face meetings. Mike Cowlishaw [1999b; Appendix I], the project editor, maintained a document that tracked the current status of sections of the specification. The status indicators were as follows: "unchanged since V1," "not ready," "discussion needed," "function accepted," and "content agreed." The status "function accepted" meant that the committee was in agreement about the functionality to be defined in the specification and the status "content agreed" meant that the actual specification text had been reviewed and accepted.

Bill Gibbons was editor of the working draft of the new specification. Each meeting had an agenda of presentations and discussions of proposals and open issues. Proposals typically were presented in the form of new or revised algorithmic specification text. Meetings also had a general status review where attendees discussed issues that they had identified since the last meeting. When there was agreement on a proposal or issue resolution, Gibbons would incorporate it into the working draft. The first complete draft for V2 [Cowlishaw et al. 1998] was released in April 1998 and was based upon the ECMA-262 Edition 1. It did not include any of the changes being concurrently developed for ECMA-262 Edition 2, the ISO edition. The working draft's title page states that it contains proposed changes submitted by Netscape and Microsoft. In September, after completion of the ISO edition, Gibbons merged the ES2 changes into the current V2 working draft.

Unicode was still a new technology and language designers were still exploring alternative approaches to integrating it into programming languages. One particular concern was how to deal with Unicode's various normalization forms which allow alternative encodings of behaviorally equivalent sequences of characters. ES1 had very minimal support for Unicode. After Tom McFarland of Hewlett-Packard attended the May 1998 meeting, he submitted a memo [McFarland 1998] identifying what he considered to be a number of issues relating to *internationalization*[g] (I18N) and better integration of Unicode into ECMAScript. In November 1998, following discussions at several meetings, TC39 established an "I18N Working Group" chaired by Richard Gillam [1998] of IBM. The I18N group quickly decided to focus on a small number of basic I18N capabilities for the core language [Gillam et al. 1999b] and to defer the more complex aspects of internationalization and localization for inclusion in a separately defined optional library [Gillam et al. 1999a,b]. However, it took until 2012 for the specification [Lindenberg 2012] of such a library to be completed. In addition to adding a small set of core language locale-specific functions, the I18N group also resolved how

- `try-catch-finally` and exception objects
- `instanceof` and `in` operators
- Object prototype methods: `hasInstance`, `hasOwnProperty`, `isPrototypeOf`, `propertyIsEnumerable`
- Global binding for `undefined`
- `toFixed`, `toExponential`, `toPrecision`
- URI handling functions
- Unicode characters in identifiers
- Basic I18N methods: Object `toLocaleString`; Array `toLocaleString`; Number `toLocaleString`; String `localeCompare`, `toLocaleLowerCase`, `toLocaleUpperCase`; Date `toLocaleDateString`, `toLocaleTimeString`

Fig. 19. New ES3 features not in pre-1998 browsers. Some of these were incorporated into browsers while ES3 was being developed by TC39.

to incorporate non-Latin characters into identifiers. It largely side-stepped the normalization issues by recommending that the ECMAScript language specification be written with the presumption that source code presented to an implementation is in Unicode Normal Form C. It also chose to not include any support for Unicode normalization in the core language and to defer programmatic support for normalization for inclusion in the optional library.

A major task for V2 was to design the exception handling mechanism for the language. In February 1998 [TC39 1998c], both Herman Venter of Microsoft and Waldemar Horwat of Netscape presented design sketches. Both designs were loosely modeled after Java's try-catch-finally statement syntax, but with significant syntactic and semantic differences from both Java and each other.

In Microsoft's design [Venter 1998b], any value can be thrown as an exception and `try` statements have a single `catch` clause that declares a local variable that is initialized to the caught exception value. Any exception propagated from the `try` block is unconditionally caught. There is no `finally`.

Netscape's design [Horwat 1998] also allows any value to be thrown as an exception. However, in this design `try` statements may have multiple `catch` clauses[38] with syntactic `instanceof` discriminators used to determine which `catch` clause to execute. If no `catch` clauses matches an exception, propagation of the exception continues up the call stack after execution of the `finally` clause. The `instanceof` discriminator was eventually replaced by an `if` discriminator[39] which evaluates an expression to a Boolean value which determines whether the descrimited `catch` is selected.

At the February 1998 meeting, the committee agreed to use the `try` and `catch` keywords and that a `throw` statement could propagate any value (not simply instances of specific built-in exception Classes) to represent an exception. At the March 1998 working group meeting [TC39 1998d], Waldemar Horwat argued for inclusion of the `finally` clause and agreed to further investigate the details of how it could be implemented. The April working draft [Cowlishaw et al. 1998] incorporated Netscape's design but issues left unresolved at that time included support for `finally`, scoping of the `catch` variable binding, whether multiple `catch` clauses would be allowed, whether `instanceof` should be used as a `catch` selector, and whether unselected exceptions should be automatically rethrown. Figure 20 provides examples illustrating the syntax used by Microsoft's proposal, Netscape's revised proposal, and what is ultimately specified in ES3. Notice that in Netscape's design a separate selector expression is used to choose a `catch` clause while in both

[38] Mike Shaver reported in 2019 personal communications that he originated the idea of having multiple `catch` clauses. Netscape's [2000] JavaScript 1.5 subsequently included multiple `catch` clauses as a non-standard extension to ES3.

[39] In some working documents [Horwat 1998; Venter 1998c] the `if` keyword that prefixed a `catch` guard expression was replaced with a colon.

Microsoft design

```
try {
  doSomething();
}catch (var e) {
  if (e == "thing")
    console.log("a thing")
  else if (e == 42)
    console.log("42")
  else {
    console.log(e);
    cleanup();
    throw e; //rethrow
  }
// no syntactic finally
}
cleanup();
```

Netscape design

```
try {
  doSomething();
}catch(e if e =="thing"){

  console.log("thing")
}catch (e2 if e2 == 42){
  console.log("42")
}catch (e3){
  console.log(e3);

  throw e3; //rethrow

}finally {
  cleanup();
}
```

Final Edition 3 design

```
try {
  doSomething();
}catch (e) {
  if (e == "thing")
    console.log("a thing")
  else if (e == 42)
    console.log("42")
  else {
    console.log(e);

    throw e; //rethrow
  }
}finally {
  cleanup();
}
```

Fig. 20. Exception Handling Alternative Designs. In these examples, the doSomething function may throw two kinds of exceptions that require distinct handling before execution continues in the current function. All other exceptions are "rethrown" to propagate them to the current function's caller. The current functions has cleanup processing that needs to be performed regardless of whether doSomething throws an exception.

Microsoft's design and the final ES3 design user logic in the single `catch` block is needed to discriminate different exceptions.

The issue of whether the language should support multiple `catch` clauses was not resolved until the final technical review [TC39 1999b] of the draft standard in September 1999 where that feature was finally deferred for future consideration. It was also only at that final review that agreement was reached on the set of built-in exception Classes that would be defined by the standard.

The `catch` clause guard expressions are an example of the difficulties the committee had with adapting features from Java and other *statically typed*[g] class-based languages to the dynamic types and prototypal inheritance of JavaScript. In Java, the determination of which `catch` clause will handle a thrown exception is made via a side-effect–free subtype inclusion test that depends upon only the statically declared class hierarchy. The test can be made before actually unwinding the call stack. But JavaScript does not have a formal concept of class or a static class hierarchy and the committee had decided to allow any kind of value to be thrown as an exception. Discriminating among arbitrary values in a JavaScript `catch` clause requires evaluating arbitrary guard expressions that potentially include assignments and function calls. But evaluation of expressions require establishing the appropriate lexical and dynamic environments and each guard expression evaluation might have side effects that could change the result of subsequently evaluated guard expressions. In one intermediate proposal, Waldemar Horwat [1998] had a complex prose specification that allowed implementations to decide when and in what order to evaluate `catch` guard expressions. It even allowed individual guard expression to be evaluated multiple times. Horwat hoped to enable debuggers to determine, prior to unwinding the stack, whether a thrown exception was unhandled. It is fortunate that this design was not accepted as subsequent experience showed that such implementation variation is a significant source of interoperability issues for Web pages that must work with multiple browsers.

Another example where TC39 had difficulty translating concepts and constructs from Java into JavaScript is the `instanceof` operator. In Java, `instanceof` is a binary operator that tests whether its left operand is an object that is an instance or a subclass instance of the class type named by its

right operand. Herman Venter's [1998a] initial proposal for `instanceof` exactly mimicked Java's syntax by requiring that the right operand be an identifier. But JavaScript does not intrinsically have the concept of classes and there are several ways to create new objects. Venter's proposal chose to assume usage of the constructor function pattern as the basis for the `instanceof` test. So, the right operand is expected to dynamically evaluate to a constructor object, that is to a first-class function value. Because the right operand is a first-class value rather than a type reference the proposal was soon generalized to allow an expression in that position. The `instanceof` runtime semantics was defined as a walk of the left operand's prototype inheritance chain searching for the object that is the current value of the right operand's `prototype` property. For many simple constructors this will produce a match for objects created by applying the `new` operator to the constructor.

New JavaScript programmers with a Java background assume that `instanceof` is a reliable way to discriminate among various kinds of objects. But many experienced JavaScript programmers avoid using it. There is no guarantee that the object returned by a constructor will pass the dynamic `instanceof` test and because of the mutability of the object meta-structures, repeated application of `instanceof` may not be idempotent. The test can also fail if the object being tested is from a different HTML frame than the constructor. Finally, even if the result is true, the object being tested still may not have the data and behavioral properties created for it by its constructor.

ES3 includes inner function declarations and function expressions similar to what was originally introduced in JavaScript 1.2. Function declarations were explicitly excluded from being nested within a `{}` block or as a substatement. Waldemar Horwat [2008b] subsequently explained why:

> 1. Hoisting such declarations to the top level (as is done with `var`) doesn't work because such functions can capture scopes that include variables that don't exist yet; ES3 didn't have local scopes but it did have exception scopes which cause the same problem. It got worse when we considered what would happen once we extended the language to have constants and dynamic (i.e. run-time) type annotations—such functions could capture uncreated constants and, worse, variables whose types hadn't been computed yet!
> 2. Binding such declarations only as they're encountered would have worked fine but we didn't want to implement this local binding in ES3 just for support of functions.
> 3. If such declarations were in the substatement position of an `if` statement, the planned intent was to create them only when the `if` expression was true (or false for an `else` clause), and put them into the nearest enclosing block-scope. This would constitute a form of conditional compilation. A block with an attribute before it would be a non-scoping block that distributes the attribute to the contained definitions, so you could attach several definitions to one `if` statement.

The major browsers ignored these concerns and went ahead and implemented function declarations within blocks. However, each implementation invented its own unique semantics for those declarations. Fifteen years later this created significant problems for the designers of ES6 [TC39 2013b, Function In Block Options; §21.3.2].

By spring of 1999, it was clear that the 3rd edition could not be completed for General Assembly approval in June but that a December approval was still possible. In March, the working group performed a triage [Clinick 1999] to identify features that would need to be cut or deferred to make the December target. The `__proto__` property, sharp variables, call objects for stack reification, and explicit closure objects were permanently cut. Features that were deferred for possible inclusion in future editions included: atomic operations, exception `catch` guards, conditional compilation, date scalars, decimal arithmetic, generic sequence operators, the optional I18N library, foreign function

interfacing, object persistence using `toSource`, support for numeric units syntax and arithmetic, and an extensible syntax for literals.

The working group had four meetings between May and September 1999 to resolve issues for the final draft of the 3rd Edition specification. Significant design issues that had to be resolved over this period included creating an algorithmic specification of regular-expression–matching semantics, deciding upon the set of built-in exceptions types, pinning down the binding semantics of function expressions, and working out the details of incorporating Unicode support into the language.

On August 8, 1999, Mike Cowlishaw [1999c] distributed the final "E3 Draft Status" showing all sections with a status of either "Content agreed" or "Unchanged since V1." On August 25, Bill Gibbons [1999] distributed the "Edition 3 Final Draft" and left the committee for a new job. Herman Venter and Waldemar Horwat took responsibility for integrating any remaining changes into the draft.

For the final ES3 development meeting [TC39 1999b], Horwat prepared a long list of notes identifying corrections for minor editorial and technical issues. There were only a few changes that had significance to everyday JavaScript programmers. The built-in exceptions `ConversionError` and `RegExpError` were eliminated and replaced by `TypeError` and `SyntaxError`.

The August draft had not specified any meaning for the optional identifier that was allowed to occur in the function name position of a *FunctionExpression*[40] such as:

```
function fact(n) {throw "wrong fact"}; //a function declaration
var lambdaFact = function fact(n) {//a function expression: does it bind fact?
  return n<=1 ? 1: fact(n-1);
};
lambdaFact(5); //should this recur or throw?
```

ES3 Draft

In that draft, calling `lambdaFact` would have thrown the exception because the name `fact` in the head of the *FunctionExpression* did not create a lexical binding for `fact`. At the September meeting there was agreement to revise the specification so that the name created a local name binding for the function that was visible only within the body of the *FunctionExpression*.

The most surprising last minute addition was a feature called "joined functions" that Waldemar Horwat proposed at the meeting. Joined function permitted implementations to repeatedly return the same function closure object in situations like this:

```
function getClosure() {return function() {/* no free variable references */}}
var firstTime = getClosure();
var secondTime = getClosure();

//It would be implementation dependent whether
//the following displays true or false
console.log(firstTime === secondTime); //the same object?
```

Only ES3

Waldemar Horwat had concerns about the overhead of closure creation, and argued that this change would allow an implementation the discretion to reuse closures in some common situations. Herman Venter expressed some concerns, but by the end of the meeting agreed to allow this change. This could have been a significant design mistake because subsequent experience with Web browsers showed that the sort of observable implementation variation permitted by this

[40] *FunctionExpression* is a non-terminal symbol of the ECMAScript grammar. By convention, such symbols are italicized.

Mike Ang	Gary Fisher	Clayton Lewis	Sam Ruby
Christine Begle	Richard Gabriel	Drew Lytle	Dario Russi
Norris Boyd	Michael Gardner	Bob Mathis	David Singer
Carl Cargill	Bill Gibbons	Karl Matzke	Randy Solton
Andrew Clinick	Richard Gillam	Mike McCabe	Guy Steele
Donna Converse	Waldemar Horwat	Tom McFarland	Michael Turyn
Mike Cowlishaw	Shon Katzenberg	Anh Nguyen	Herman Venter
Chris Dollin	Cedric Krumbein	Brent Noorda	George Wilingmyre
Jeff Dyer	Mike Ksar	Andy Palay	Scott Wiltamuth
Brendan Eich	Roger Lawrence	Dave Raggett	Rok Yu
Chris Espinosa	Steve Leach	Gary Robinson	

Fig. 21. Technical Contributors to ECMA-262 Editions 1, 2, and 3.

feature could prevent websites from successfully working on all browsers. Fortunately, no browser implemented the joined functions feature and in 2009 it was removed, from the ES5 specification.

Usage of octal constants (written with a leading `0` digit) and octal escape sequences in string literals were discouraged by moving them from the *normative*[g] specification into the non-normative Annex B[41] of the standard. Also moved to Annex B were the non-Y2K compliant Date methods, the `escape` and `unescape` string functions, and the String method `substr`. These were all features that were considered obsolete but were still used by websites. The hope was that listing features in the standard's non-normative Annex B would signal that these were deprecated features, which should not be used and that implementations were empowered to eventually remove them. This was a naïve expectation. TC39 members did not yet appreciate that browser implementors would be extremely reluctant to remove any feature, whether standardized or not, that may have actually been used on Web pages—some Web pages never go away.

After reviewing and resolving all of the outstanding issues, TC39 accepted without objections the specification as complete, subject to incorporating the changes from the meeting. Waldemar Horwat and Herman Venter prepared the final document [TC39 1999e] and passed it to the Ecma Secretariat on October 13, 1999. The final draft included a list (Figure 21) of everyone who contributed to the first three editions of ECMA-262 by authoring content, attending a technical meeting, or contributing via email.

In November, several minor editorial and technical errors in the final draft were identified and corrected [TC39 1999a]. Most significantly, Microsoft had discovered that a number of websites (including microsoft.com) broke when they changed JScript's implementation of `String.replace`, using a regular expression, to conform to the final draft. TC39 agreed to change the specification to match Microsoft's previous implementation.

On December 16, 1999, the Ecma General Assembly [Ecma International 1999] approved the specification as *ECMA-262, 3rd Edition* [Cowlishaw 1999a]. Starting in March 2000, Waldemar Horwat [2003b] maintained an unofficial ES3 errata. The major browsers shipped ES3 compliant versions over the course of 2000 with Microsoft's JScript 5.5 shipping as part of IE 5.5 in July 2000 and Netscape shipping JavaScript 1.5 as part of Netscape 6 in November 2000. *ECMA-262 3rd Edition* was not replaced by a newer edition until December 2009. During that period, browsers were not automatically updated and many users updated their browser only when they got a new computer or new version of their operating system. It would be nearly a decade before Web developers could assume that all users would be using a browser that supported ES3 features.

[41] Annex B is an appendix to the ES3 specification which provides definitions of obsolete ECMAScript features.

13 INTERLUDE: JAVASCRIPT DOESN'T NEED JAVA

Originally JavaScript was conceived as a Java side-kick scripting language and all sophisticated programming tasks would be done using Java. But as experience with JavaScript grew, Web developers began to realize that all they really needed was JavaScript.

13.1 The Evangelist

As JavaScript usage in browsers grew, JavaScript educators and evangelists emerged. One of the most influential was Douglas Crockford. Starting with a short online essay titled "JavaScript: the World's Most Misunderstood Programming Language" [Crockford 2001a], he undertook the task of changing how the software development community perceived JavaScript. In another essay Crockford [2001e] explained:

> When JavaScript was first introduced, I dismissed it as being not worth my attention. Much later, I took another look at it and discovered that hidden in the browser was an excellent programming language. My initial attitudes were based on the initial positioning of JavaScript by Sun and Netscape. They made many misstatements about JavaScript in order to avoid positioning JavaScript as a competitor to Java. Those misstatements continue to echo in the scores of badly written JavaScript books aimed at the dummies and amateurs market.

Douglas Crockford [2001d; 2002a; 2003; 2006] exposed JavaScript's Scheme-like closures and Self-like objects and explained how to use them. But he did not gloss over JavaScript's warts and quirks. In addition to identifying them, Crockford [2001e; 2002d] created and promoted JSLINT [Crockford 2001b], the first widely used linter[42] utility for JavaScript. Crockford [2001c; 2019b] also introduced the concept of minimization[43] to JavaScript developers and created the JSMIN utility. He wrote a best-selling book [Crockford 2008b] that told programmers how to use JavaScript's good parts and avoid the bad parts. Eventually, he became a participant in JavaScript standardization efforts.

Crockford championed simplicity and realized that the complexities of XML could be avoided by using a subset of JavaScript's object and array literal syntax as a language-independent data interchange format. He named this widely adopted format "JavaScript Object Notation" or "JSON" [Crockford 2002b,c; Crockford 2019a]. This simple format could be easily parsed in any language but was particularly easy to deal with in JavaScript where the `eval` function could convert a JSON data record into JavaScript objects.[44]

13.2 Rich Internet Applications and AJAX

Early interactive Web applications were primarily forms based. Users would enter data into HTML forms that were transmitted by a browser back to a Web server where the data would be processed and databases updated. An updated HTML presentation was then transmitted back to the browser for display. JavaScript was used browser-side for basic input data validation and simple dynamic changes to the sever generated HTML. This style of Web application was subsequently characterized as Web 1.0.[45]

Some applications are highly interactive requiring a rich low-latency user interface. It was inevitable that some developers would want to develop Web applications with those characteristics.

[42] A development-time tool that checks source code for dubious coding practices and error-prone constructs.
[43] Mechanically reducing the download size of JavaScript program by removing comments, unnecessary whitespace, and performing other semantic preserving source code transformations.
[44] It was ultimately recognized that using `eval` to process JSON was a security hazard that exposed applications to code injection attacks. Modern JavaScript engines use dedicated JSON parsers which are not susceptible to such attacks.
[45] DiNucci [1999] made early use of the terms Web 1.0 and Web 2.0.

In 1995, when Netscape introduced both Java and JavaScript into its Web browser, the plan was that Java would be the primary language for implementing complex interactive Web applications and JavaScript would primarily be used within forms-based applications [Shah 1996]. During the late 1990s and early 2000s many "Rich Internet Applications" [Allaire 2002] were built as Java Applets.

In 1997, Microsoft released a Web-client version of its enterprise email client. Outlook Web Access (OWA) [Bilic 2007; Van Eaton 2005] was implemented as a Web-1.0–style application. OWA 1.0 was succeeded by a richer version that used Dynamic HTML[46] and a new browser API called XMLHTTP [Hopmann 2006]. XMLHTTP enabled JavaScript code on a Web page to asynchronously transfer data to and from a server without completely reloading the Web page. The combination of DHTML and XMLHTTP allowed a Web page to load once per session and then operate as an interactive application with remote access to data and services.

Over the course of the first half of the 2000s various organizations built Web applications using these and similar techniques. But this Web application style did not become widely known until Google used it to implement GMail, Google Maps, and other applications. Jesse James Garrett [2005] coined the term "AJAX" to describe it. AJAX and the social media applications built using it became the hallmark of the *Web 2.0*[g] era.

The emergence of Web 2.0 and AJAX was a major transition point for the role of JavaScript in Web development. JavaScript's role was changing from a language for adding dynamic elements to mostly static pages to a language for coding complex Rich Internet Applications.

At the same time, the browser ecosystem was growing more complex. There had always been a variety of alternative browsers with very low market share. The gradual abandonment of active browser development by Netscape (after its acquisition by AOL) and by Microsoft (after it achieved market dominance) created the opportunity for new browsers to emerge. *Firefox*[g][47] [Mozilla 2004], *Opera*[g] [Opera 2013], Apple *Safari*[g] [Melton 2003], and eventually Google *Chrome*[g] [Kennedy 2008] gradually gained meaningful market share.

The new browsers all implemented their interpretation of the ES3 JavaScript specification and the browser platform APIs that were partially specified by the W3C. But the platform specifications were incomplete or imprecise. Most of the new browsers extended or modified the platform APIs in various ways. And while these new browsers were emerging, many users continued to use outdated versions of Internet Explorer and Netscape that were buggy and lacked support for the latest language features and platform APIs.

Web browsers are different from most other application platforms in one important respect—applications are distributed in source code form for immediate execution in an environment provided by the user of the application. This differs from the more traditional scenario where a developer may choose a specific version of a compiler and runtime library and then build and test their application before deploying it in binary form to users. Douglas Crockford,[48] in various talks, characterized this aspect of Web development as: the user chooses (usually without knowledge) the language processor. Web developers needed to ensure that their Web pages and Web applications worked correctly on whichever browser an end-user chooses.

One way to deal with browser differences is to create a separate version of an application for each incompatible browser. The Web server can then send different versions to different browser based upon identifying information provided by the browser when requesting a Web page. But typically most of the application source code is shared by all versions with only small variations to

[46] Dynamic HTML (DHTML) uses JavaScript to dynamically modify the HTML elements of the active Web page.
[47] An outgrowth of Netscape's Mozilla Project where Brendan Eich was lead architect.
[48] personal communication 2019

account for the browser differences. This creates the development and operational challenge of maintaining multiple, mostly identical, versions of the application.

A way to avoid multiple distinct versions of the application source code is to have a single source file where the browser-specific variations are dynamically selected as the application runs within a browser. Variations are selected using idiomatic code sequences that perform browser sniffing (identifying a specific browser version) or feature testing (identifying the presence of specific features or bugs).

The complexity of AJAX-style applications combined with browser interoperability issues led to the emergence of application frameworks and libraries to simplify Web application construction. Early frameworks included Prototype [Stephenson et al. 2007], MooTools [Proietti 2006], and Dojo [Russell et al. 2005]. The one with the greatest adoption [W^{3}Techs 2010] was jQuery [Resig 2006]. These early frameworks and libraries typically provided a structure for AJAX-style applications along with higher-level abstractions that simplified the coding of tasks commonly performed by such applications. They also resolved many of the interoperability issues by internally dealing with and hiding many of the browser functional variations.

One specific kind of library was significant enough that a new word was coined to characterize it. The term "polyfill[g]" was coined by Remy Sharp [2010] to describe a library that provides APIs that should be provided by a browser but which were missing. A well-designed polyfill dynamically checks whether the feature it provides is already available and installs itself only if built-in support is absent or incompatible. Early polyfill libraries focused on making browsers more interoperable by hiding legacy feature variations that carried forward from early browser competition or supporting new browser features in old browsers. If a feature was present in one popular browser and absent from others, a polyfill could enable a Web application to use the same code to operate on all browsers. As browser interoperability improved, polyfills became common as a way to provide early access to new browser and JavaScript functionality. It became common to create a polyfill library as part of the design process for new features. In addition to being useful for developers, polyfill usage generated valuable developer feedback on new functionality and API designs.

Naming clashes were common when JavaScript applications were created by naïvely composing independently created pieces. Many frameworks and libraries provided some sort of modularity mechanism typically constructed using namespace objects and immediately invoked function expressions (IIFE[49]). A namespace object is simply a singleton object whose primary purpose is to provide qualified name access to functions or variables. JavaScript 1.0's built-in `Math` object is a namespace object. A limitation is that all of names in the namespace are public. That limitation can be overcome by combining a namespace object with an IIFE in the module pattern as illustrated in Figure 22.

The module pattern has several variations, but the basic concept is that the lexical scoping of an IIFE (or sometimes a named function) is used to encapsulate some private state with a set of functions. The IIFE returns a namespace object whose properties are the encapsulated functions that need to be publicly accessible.

Douglas Crockford is often credited with popularizing the module pattern but it was likely independently discovered by many JavaScript programmers.

13.3 Browser Game Theory

During the *browser wars*[g] [Borland 2003], Netscape and Microsoft tried to out-innovate each other in the introduction of new website capabilities. They both tried to convince developers to use

[49]Immediately invoking a function is a substitute for a block-scope. This technique was known to Scheme programmers and was widely used by JavaScript programmers starting in the mid 2000s. Ben Alman [2010] coined the term IIFE.

```
//define services using the module pattern
var Services = function () {
  var privateJobCount = 0;  //private variable of the "module"
  return {  //the namespace object
    jobCount: function {return privateJobCount},
    job1: function() {this.jobCount++}
  }
}(); //Services is initialized to the result of calling the function

//access entities from the namespace
Services.job1();
console.log(Services.jobCount());  //should display 1
```

ES3

Fig. 22. An example of the JavaScript module pattern. The `Services` function encapsulates a private implementation. `Services` is initialized when called and returns a namespace object whose properties exposes the public interface of the "module'."

their unique features and ran "Works best on [XXX]" marketing campaigns. But browser users got annoyed when websites did not work correctly with their preferred browser and Web developers did not like having to code multiple versions of their sites for different browsers.

Even while Microsoft was investing heavily in both technical and non-technical means to win market share from Netscape there was recognition that evolution of JavaScript would require collaboration in addition to competition. At the July 1997 TC39 meeting, as work on the first edition of ECMA-262 was nearing completion, Microsoft's Scott Wiltamuth presented a pledge of cooperation (Figure 23) regarding future ECMAScript development.

Brendan Eich recalls that at some point he realized that market pragmatics severely constrained what browser implementors could do to improve their products, for example:

- Breaking-changes (even bug fixes) can drive away users.
- New browsers must conform to what is already there.
- Innovation is wasteful if only present in one browser.
- First browser to try something new may actually lose market share.

Eich recognized that this was likely a Nash Equilibrium [Nash 1950] situation and coined the term "Browser Game Theory" to describe the constraints that browser implementors operate under.

The first constraint is sometimes expressed using the slogan "Don't break the Web!" Web pages are typically stored on servers as HTML and JavaScript source code that is reinterpreted by a browser each time a user access the page. Many of these pages are unmaintained by their original creators but still actively used. Some are documents with ongoing utility or historical importance. A *breaking-change*[g] to how a browser interprets the source code can cause a page to become illegible or non-functional. If the change occurs only on a single browser, users may switch to using different browsers. If such a change is pervasive among browsers, parts of the unmaintained Web become permanently broken. This fact also limits the developers of Web standards. A standard that introduces a new feature or mandates a change will be ignored by browser implementors if they believe that it will invalidate significant existing Web content.

Today browser developers generally understand that the interoperability requirements of the Web and its open standards foundation limit their ability to compete via unilateral platform innovations. Browsers can and do compete on quality of implementation grounds such as performance, security,

A different way of working
Microsoft's ECMAScript standards pledge
- We will bring new ideas that impact ECMAScript to the group's attention, as opposed to keeping them secret.
- We will implement ideas that have achieved consensus in the group.
- We will follow the architectural principles guiding the group, rather than release alternatives which ignore or contradict these principles.
- We will not ship ECMAScript extensions without first submitting them to ECMA.
- We will implement all ECMA-approved ECMAScript standards.
- We will clearly identify any not-yet-approved ECMAScript features that we support as such.

Fig. 23. Microsoft Pledge at July 1997 TC39 Meeting [TC39 1997g]

reliability, and usability. But advancing the basic technical capabilities of the browser application platform usually requires cooperation among all the major browsers.

Browser game theory was a significant factor in the evolution of JavaScript. Moreover, it can provide a perspective for understanding why JavaScript became successful and an explanation for many of the innovation successes and failures that occurred over the course of its history.

Part 3: Failed Reformations

14 DISSATISFACTION WITH SUCCESS

As the end of the 1990s neared, it was clear that the Internet and in particular the World Wide Web was having a phenomenal impact upon the world [Miniwatts Marketing Group 2019]. The rapid growth of the Web had been enabled by the incremental pragmatic enhancement of browser technologies by Netscape, Microsoft, and other browser developers. The success of the Web and the necessity of coordinating the ongoing evolution gave rise to standards groups such as Ecma TC39 and W3G working groups. Some of the participants in those groups were subject matter experts who were not directly involved with browser development. Their interest was focused on an idealized future Web. From that perspective, the existing pragmatically developed Web technologies were viewed as an impediment to that future.

In May 1998, the W3C held a workshop titled: "Shaping the Future of HTML." The conclusions in the record of the workshop say:

> In discussions, it was agreed that further extending HTML 4.0 would be difficult, as would converting 4.0 to be an XML application. The proposed way to break free of these restrictions is to make a fresh start with the next generation of HTML based upon a suite of XML tag-sets. The workshop expressed a need for a better match to database and workflow applications, and for the widely disparate capabilities of small/mobile devices. Modularizing HTML will provide the flexibility needed for this. [W3C 1998]

David Singer [1998], representing IBM, was more blunt in a workshop presentation: "The Future of HTML as we know it should be: Nasty, Brutish, and Short."

As ES3 approached completion, TC39 found itself in a similar situation. With ES3, ECMAScript had caught up with the JavaScript features provided by the Netscape and Microsoft browsers and, at least initially, the browser vendors weren't providing much guidance regarding what to do next. Unlike Netscape in 1995, TC39 was not constrained to avoid Java-like capabilities. Some TC39 participants saw a need for a second-generation browser scripting language that corrected

mistakes made in the original JavaScript design and that offered features [Raggett 1999b; TC39 1999c; Appendix J] catering to the needs and sensibilities of professional software developers rather than non-professiional script writers. This new generation of ECMASript was targeted to be the 4th edition of of ECMA-262. Within TC39, it was initially called "E4" and later "ES4."

15 ES4, TAKE 1

From the very first TC39 meeting—where Borland International [1996] presented a proposal for adding class definitions to the language—there had been interest in adding features to JavaScript to help manage the complexity of larger programs. Netscape's JavaScript 1.2 supported cryptographically-signed scripts which integrated with each other via `import` and `export` declarations [Netscape 1997a]. Microsoft's JScript 3 included its conditional compilation features [Clinick 1997]. The February 1998 version of the ECMAScript Futures List [TC39 1998c] lists "package concept" as a possible item for V2. Such programming-in-the-large features were relatively quickly dropped from the ES3 feature set but work on them continued in parallel within TC39.

The first major proposal came from Dave Raggett who was a W3C Fellow sponsored by Hewlett-Packard. At the W3C, Raggett was developing a proposal named "Spice" to improve the integration of HTML, CSS, and JavaScript. An early version of the proposal [Raggett 1998c] was submitted to TC39 in February 1998. In addition to HTML and CSS integration features, Raggett's initial proposal included a construct for declaring prototype objects which was similar to the Borland class declaration proposal. It added the ability to declaratively associate event handlers with prototype objects. The proposal also included constructs for defining "libraries" and for importing definitions from libraries, as follows:

```
import document, block, Inline from "http://www.w3.org/Style/std.lib";

prototype Link extends Inline
{
   href = "http://www.w3.org/";
   when onmousedown
   {
      document.load(this.href);
   }
}
```

February 1998 Spice Proposal

The March 1998 meeting notes [TC39 1998d] show that Dave Raggett's initial Spice submission was discussed and remarked that the "initial feedback is negative." Raggett continued to evolve his proposal, working with Chris Dollin and Steve Leach, two language designers from HP Labs. In September, Raggett submitted a new set of documents [Raggett et al. 1998] describing an expanded Spice proposal. The proposal was, in effect, an incompatible replacement language for ECMAScript—it even replaced curly-brace delimited C-style syntax with a closing-keyword-based statement syntax.

Dave Raggett [1998a] presented the revised Spice proposal at the November 1998 TC39 working group meeting. This meeting had been preceded earlier in the month by a private meeting between the Spice designers and TC39 delegates from Netscape and Microsoft. At the working group meeting, TC39 members had no interest in replacing the then-current statement syntax or in immediately trying to integrate declarative support for style sheets. However, there was interest in extending ECMAScript with some of the concepts in the Spice proposal such as classes, numeric units,[50]

[50]Numeric units means annotating numeric values with units of measure such as meters and kilograms. For Web pages units such as pixels and points were of particular interest.

```
// C-style declaration alternative
var float x, int[] y, z;  // what is type of z?
var float x, int[] y, int[] z; //is it this?
var float x, int[] y, any z;  //or this?

// Pascal-style alternative
var x: float, y: int[], z;  //type of z is any
```

Fig. 24. Early ES4$_1$ discussions considered both C and Pascal derived syntax alternatives for typed declarations

types, and modules. When asked, Raggett indicated that it was unlikely that HP would continue to develop Spice if comparable features were added to ECMAScript.[51]

A new TC39 Spice Working Group was chartered to develop a proposal for presentation in January 1999 to the full group. The sense of the committee was that new features in support of new core concepts would have to be defined using the already reserved Java keywords and that the semantics of classes should be similar to Java. Numeric units should be defined on top of classes and would require adding operator overloading.

The Spice working group's initial teleconference occurred in the first week of December 1998. On December 10, Dave Raggett [1998b] distributed a new document based on that meeting. It touched upon packages and numeric units but more extensively explored type declarations including class and interface definitions. Its focus was more on syntax than semantics. The document assumes a *nominal type system*[g] with named built-in primitive types, a homogeneous array type, class types where subclasses are nominal subtypes, interface types, and an `any` type which indicates that accesses need to be dynamically type checked. Syntactically it explored alternatives for associating types with variable bindings. It assumed that the `var` keyword would still be used for variable declarations and explored both the C-style of using a type expression as a prefix to a declared name and the Pascal-style of using a colon and type expression following a declared name. The alternatives are exemplified in Figure 24.

Class and interface definition syntax roughly followed Java and included the full complement of `public`, `private`, `protected`, and default (package) visibility modifiers. The underlying metaobject structure was not covered but it was implicit that the metaobject model would have to be different from the then-current JavaScript prototypal inheritance model. The document raised issues about distinguishing between early bound member access using declared static type information and late bound member access where no static type information is available. Dynamically adding properties[52] was explored and the document suggested that it might be desirable to permit a class to disallow them.

Design discussions, primarily related to classes, *type annotations*[g], and scoping, continued in January and February 1999 [Raggett 1999b,c] with Chris Dollin, Waldemar Horwat, and Herman Venter being the primary participants. Much of the discussion concerned the nature of class-defined objects and the semantics of class member access. Dollin and Venter generally preferred a Java-like semantics where the structure of a class instance is statically determined by the class declaration, and member accessibility is statically determined based upon type information available at the referencing site. Horwat generally favored a more dynamic model where even in the presence of type annotations, fallible dynamic lookup is used to access members. The optional type annotations,

[51] Chris Dollin and Steve Leach continued to develop a Spice language that was not based upon JavaScript [Dollin 2002], and Leach subsequently evolved it into the Ginger programming language [Leach et al. 2018].
[52] Microsoft called these "expando properties."

- Modularity enhancements: classes, types, modules, libraries, packages, etc.
- Internationalization (I18N) items:
  - Internationalization library [possibly as a separate ECMA technical report]
  - Calendar
- Decimal arithmetic (enhanced or alternative Number object)
- Catch guards (with types)
- Atomic (thread-safe) operations discussion/definition (possibly non-normative)
- Miscellaneous enhancements [other than modularity]:
  - Declaration qualifiers extension mechanism
  - Extensible syntax (e.g., use of # for RGB values)
  - Units syntax and arithmetic library
  - "Here documents" (long string constants)

Fig. 25. Provisional ES4$_1$ features from TC39 November 1999 "Futures List" [TC39 1999d]

expando properties[g], the expectations of existing JavaScript programmers, and compatibility with existing code that used prototype-based ad hoc classes all seemed to require a more dynamic semantics. In addition, Horwat argued that a dynamic semantics was more aligned with the nature of scripting which involves the dynamic assembly of code from multiple sources and use of libraries that version independently from the scripts that reference them. Horwat [1999b] summarized the difference between the static and dynamic approaches in a document describing member lookup alternatives.

At the February meeting Waldemar Horwat [1999a] revealed his specification for "JavaScript 2.0." He characterized it as an experimental design that had originally been written for Netscape[53] which matches much of what had been recently discussed by TC39.[54] It included a nominal type system with a large set of machine-level numeric types, Java-like class member visibility rules, and packages with explicit imports. It also had a number of more novel features including class extension declarations, declaration-level versioning of package members, nullable and non-nullable types, and first-class type values. Instead of the declaration-hoisting semantics of previous JavaScript versions, JavaScript 2.0 proposed a "streaming execution model" [Raggett 1999d] where declarations are not processed until encountered during execution. For example an `if` statement could be used to conditionally declare a variable or to select between declarations with differing type annotations. The combination of first-class type values and streaming execution of declarations made full static type checking impossible in some cases.

JavaScript 2.0 was not an attempt to be fully backward compatible with original JavaScript or even with the still-not-completed ECMAScript 3. When introducing JavaScript 2.0 to TC39, Waldemar Horwat said: "At a bare minimum you should be able to write code that works in ECMAScript 1.0 and 2.0 [ES4]. Full backwards [*sic*] compatibility would be rather painful." [Raggett 1999c] For example, the syntactic complications of the optional type annotations precluded supporting automatic semicolon insertions on line breaks. Horwat's solution to backward compatibility was for implementations to provide multiple compilers. He believed that switching compilers according to the language version was preferable to a single language with strict forward compatibility.

Much of TC39's attention for the remainder of 1999 was focused on completing ES3, but in March it produced a "Futures List" [TC39 1999c] of possible post-ES3 features. The Spice Working Group transformed into the Modularity Subgroup and continued to occasionally meet [Raggett

[53] Mozilla's source code repository contains source code [Horwat et al. 2005] of *Epimetheus* [Horwat et al. 2003], Netscape's experimental JavaScript 2 implementation.

[54] Venter believes (2018 personal communication) that Raggett's proposals had little influence on Horwat's design.

1999a,d; TC39 1999a] regarding $ES4_1$. The pace picked up in November when TC39 shifted its primary attention to "Edition 4" and updated the post-ES3 futures list (Figure 25). The November 1999 report of the TC39 chair [Lewis 1999a] describes the goal of $ES4_1$:

> ECMAScript 2.0 [$ES4_1$], an ambitious much-improved ECMAScript language definition that [the committee] hopes to standardize in the year 2000 (though this may be overly ambitious). The principal goal of ECMAScript 2.0 is to provide support for 'programming in the large' - that is, to support construction of programs written by several different people and assembled, perhaps for the first time, on the user's desktop.

At the January 2000 meeting [Raggett 2000], Microsoft pushed for a December 2000 publication date for the 4th Edition by trimming features to meet that date. Microsoft's primary interests were the addition of static type annotations and maintaining backward compatibility, including support for Automatic Semicolon Insertion. Venter circulated a set of changes to the ES3 specification that he believed would be sufficient for supporting type annotations. However, there remained much uncertainty about the nature of the type system, the semantics of classes, packages, and name spaces, and how to integrate static and dynamic language concepts into a single language.

On June 22, 2000, Microsoft [2000b] announced the .NET Framework. This was Microsoft's competitive response to Sun's Java platform. Microsoft's .Net was a multi-language application-development platform. In addition to its premier language, C#, it supported dialects of Visual Basic, JavaScript, and other languages. The announcement was followed in July by the release of the first .NET preview build[55] at Microsoft's Professional Developer's Conference [Microsoft 2000a]. The preview included an early version of JScript.NET [Clinick 2000]. Unlike JavaScript in browsers, JScript.NET was a precompiled language that targeted the .NET Common Language Runtime (CLR) and used the .NET type system internally. Internet Explorer did not support JScript.NET (or .NET in general); instead, JScript.net could initially be used to construct desktop, server, and command line applications using various .NET framework components. JScript.NET claimed compatibility with the ES3 specification, but because it was not expected to run JavaScript code written for browsers, strict backward-compatibility was not a significant concern. In addition to ES3 features, JScript.NET added optional static type annotations, class and interface declarations which included member visibility attributes, and packages with explicit imports. Microsoft's Andrew Clinick [2000] wrote that the new features had been designed in conjunction with other Ecma TC39 members and warned that details of the design might change based upon ongoing TC39 discussions.

Prior to the June 2000 .Net announcement, Microsoft's Herman Venter was unable to discuss .NET or JScript.NET with Waldemar Horwat and other TC39 members. In August, Horwat and Venter met privately to try to find alignment that would enable the completion of an ES4 standard. Horwat's [2000] meeting notes record discussion of 43 issues or points of disagreement. It summarizes the discussion as follows:

> General: Herman [Venter] is readying an implementation of JScript for servers and wants to freeze the language and make it easily interoperable with Microsoft's .NET runtime. Waldemar [Horwat] is concerned about the language's applicability to browsers and retaining the language's dynamism, which he sees as the language's differentiator. He's concerned about the language's drift towards Java or C# because he thinks that there is little need for another language in that space, and the result would be inferior to C# for static programming anyway. Herman also recommends that new server projects use C# instead of JScript, seeing the new JScript as a language geared towards developers already used to programming in JScript.

[55] The .NET Platform 1.0 shipped on February 13, 2002.

Horwat [2003a] forked a separate "ECMAScript 4 Netscape Proposal" document from the JavaScript 2.0 document. This document was then used as the working draft for ongoing $ES4_1$ development. The JavaScript 2.0 document continued to be maintained in parallel including additional features that TC39 had not agreed to include.

Microsoft wanted .NET and its languages to be perceived as standards-based technologies. Ecma had a reputation as an organization where it was easy to move proprietary technologies onto a standards track and Microsoft was happy with how TC39 had worked out. So it proposed to Ecma that the scope of TC39 be expanded and that .NET be standardized within it. TC39 was rechartered as the Ecma technical committee for "Programming Environments." The ongoing ECMAScript activity was demoted to Task Group status within TC39 and become known as TC39-TG1. Additional TC39 task groups were formed for developing standards for the CLR and for C#.

Work on this attempt to create a 4th Edition of the ECMAScript specification would continue for another three years, but in retrospect the announcement of JScript.NET was the beginning of the end of the effort. By June 2000, Netscape had lost the "Browser Wars" [Borland 2003] with its browser market share having dropped below 14% [Reuters 2000]. After being acquired by America Online it was losing staff, operating with reduced resources, and struggling to ship new versions of its browser.

Microsoft, with Internet Explorer, had won and ultimately achieved over 90% market share. It had little ongoing interest in enhancing the Web-programming platform over which it had no proprietary control. Internally, resources were redirected from enhancing open browser technologies such as ECMAScript to developing proprietary Microsoft technologies such as the Windows Presentation Framework[56] [Microsoft 2016], which it hoped would ultimately obsolete and displace open Web technologies. In the area ofprogramming languages for .NET, it focused on C# and VisualBasic.NET. In that context JScript.NET was relevant only to the degree it enabled JavaScript programmers to migrate to the .NET platform.

TG1 continued to meet, discuss specific issues, and update the draft specification.[57] There was significant ongoing disagreement between Microsoft and Netscape regarding the nature of the type system. Waldemar Horwat presented a paper [Horwat 2001] on the design of JavaScript 2.0 at the MIT Lightweight Languages Workshop where he characterized JavaScript 2.0 as having "Strong Dynamic Typing." He further explained that in JavaScript 2.0, all variables have an associated type that limits the values that can be stored into them, but the checking of the type constraints must occur at runtime. JavaScript 2.0's first-class type values and implicit downcasts[58] makes it impossible, in the general case, to statically type check its programs.

The frequency of TG1 meetings and attendance gradually decreased. Chris Dollin attended for the last time in June 2001. Herman Venter's last TC39-TG1 meeting was June 2002. On July 15, 2003, America Online announced that it was disbanding Netscape and laying off most of its employees, including Waldemar Horwat. At a TG1 meeting held that same week, Horwat resigned as ES4 editor. The remaining members of TG1 decided to focus their effort on developing XML support for ECMAScript and to suspend work on ES4 until after the XML project was completed and a new editor could be identified.

16 OTHER DEAD-ENDS

In the mid- to late-1990s there was significant interest in the concept of software components and several software component models were proposed and implemented. These included CORBA

[56] Later rebranded as Windows Presentation Foundation.

[57] The JavaScript 2 Web page [Horwat 2003c] contains the specification change log from February 1999–June 2003.

[58] A downcast checks that the value of a variable declared with some type is usable in a context that requires a value of a more specialized subtype.

from the Object Management Group (OMG), Microsoft's COM, and Sun's JavaBeans. Generally, a software component model was a modularization scheme that provided a way to describe, discover, and consume object-based software modules. At the July 1997 meeting of TC39 [1997g] Jim Tressa, representing Oracle, made a presentation about an OMG RFP for a component scripting language. At that meeting it was reported that IBM, Netscape, Oracle, and others were interested in responding with an ECMAScript-based proposal, but the specification ultimately produced by the OMG was not based upon ECMAScript.

ECMAScript Components was an attempt to promulgate a JavaScript-specific component model for use in browsers and other JavaScript hosts. It specified an XML schema and vocabulary for describing JavaScript components and a set of implementation conventions. The sponsors of the effort were NetObjects, Inc.[59] and Netscape. Richard Wagner [1998] of NetObjects made an initial presentation to the Ecma GA in June 1998. At the same meeting a draft technical specification [Wagner and Shapley 1998] was submitted to TC39. That document evolved through three more drafts and was submitted to the Ecma GA. It was approved as an Ecma standard and published as ECMA-290 [Wagner 1999]. There is no record of that standard being implemented. Based upon TC39's recommendation, the Ecma GA voted in 2009 to withdraw ECMA-290 as a standard [Ecma International 2009b].

The ECMAScript 3rd Edition compact profile project defined a language *profile*[g] for a less-dynamic subset of ES3 that would allow JavaScript implementations for resource-constrained environments to still claim conformance to the ECMAScript specification. Its creation was motivated [Raggett 2000] by an effort, WMLScript, outside of Ecma to define a JavaScript dialect for use in cell phone applications[60] [Lewis 1999b]. The Compact Profile included all features of ES3 but it permited implementations to exclude support for the `with` statement. Implementations could also exclude `eval` and the `Function` constructor. The Compact Profile also permitted an implementation to make the objects of the built-in library immutable and hence allowed for the possibility of precompiled or ROM-based implementations. The Ecma GA approved the Compact Profile standard as ECMA-327 [Vartiainen 2001]. Unlike ECMA-290, ECMA-327 was actually implemented for some environments. But as new editions of ECMA-262 were released, there was a lack of interest in updating ECMA-327. Recent editions of ECMA-262 have been implemented for very resource-limited environments. If an implementation for such environments needs to exclude certain features, they simply do so. In practice, perfect JavaScript interoperability among implementation has not proven to be a requirement for most resource-constrained applications. The Ecma GA voted in 2015 to withdraw ECMA-327 as a standard [Ecma International 2015b].

In 2002, TC39-TG1 shifted most of its attention to developing an "ECMAScript for XML" specification. E4X was a separate Ecma standard that added syntactic extensions to ES3 to support the processing of XML documents. Editions of ECMA-357 [Ecma International 2004; Schneider et al. 2005] were issued in 2004 and 2005. Firefox was the only browser to implemented E4X and hence, consistent with Browser Game Theory, it was seldom used. In 2015, ECMA-357 was withdrawn as an Ecma standard because the E4X extensions were incompatible with ECMAScript 2015 [Ecma International 2015b].

17 FLASH AND ACTIONSCRIPT

Macromedia's *Flash*[g], later acquired by Adobe, emerged in the early 2000s as a popular alternative to both Java and JavaScript for constructing Rich Internet Applications. Flash was originally a timeline-based animation product based upon work by Jonathan Gay [2006]. Flash consists of

[59] NetObjects was an IBM-funded startup.

[60] Cell phones of that era had very limited processor, memory, and communications bandwidth resources.

a visual authoring tool that compiles animation-based applications into binary files, which are interpreted by the Flash Player. The player component was integrated into browsers using the browsers' plug-in extension APIs. At its high point, a Flash player was installed by practically all browser users [Adobe 2013].

Initially Flash authoring was primarily visual, but it also included the ability to write short textual "actions" to define responses to various timeline events. In Flash Version 4, released in May 1999, Gary Grossman had evolved Flash actions into a simple dynamically-typed scripting language with syntactic similarities to JavaScript. With the release of Flash 5 in 2000 the scripting language became a dialect of ECMAScript 3 and was named "ActionScript." *ActionScript*[g] 1.0 supported most ES3 statement forms and prototype-based objects but lacked support for regular expressions, had a non-standard `eval` function which could evaluate only a restricted set of variable access expressions, and had various other subtle semantic differences. Because ActionScript code was compiled to run exclusively in the Flash Player environment, strict semantic conformance to the ECMAScript specification was not a concern. For example, in ActionScript 1.0, `var` declarations are scoped to the closest enclosing block rather than being scoped to the enclosing function.

ActionScript 2.0 was introduced in 2003 as a component of the Flash MX development environment and Flash Player 6. ActionScript 2.0 extended ActionScript 1.0 with class declarations, interface declarations, type annotations on declarations, and an import statement for accessing classes defined in other scripts. The syntax for class annotations, class declarations, and interface declarations roughly follows that used in the draft $ES4_1$/JS2 specification, but with a greatly simplified semantics. The use of type annotations was optional. Type checking was "a compile-time-only feature" [Macromedia 2003]. Java-like nominal type checking was performed at compile-time when type annotations were provided. Type information was erased before code was generated. ActionScript 2.0 uses the same virtual machine as ActionScript 1.0 and performs basic runtime safety checks. Programs can dynamically modify objects in a manner that violates the rules of the nominal type system as long as such changes do not trigger any of the runtime safety checks.

In 2003, the wide adoption of Flash for Web development was leading to the creation of large and complex ActionScript applications and some of them were encountering performance issues. Like most ECMAScript language designers and implementors at that time, the Macromedia team believed[61] that dynamic typing (particularly of primitive types) was the main performance bottleneck, and were exploring ways to add static typing to the ActionScript runtime. Around this same time, Jeff Dyer, who had been a TC39 delegate since 1998, joined Macromedia. Dyer confirmed that TC39 shared that same perspective about static typing. This widely shared view of static typing in virtual-machine-based languages was strongly influenced by the design of the statically typed Java Virtual Machine (JVM). Jonathan Gay's and Lee Thornason's Maelstrom project was a Macromedia experiment to see if a JVM could be integrated into Flash and used as the runtime for a statically typed version of ActionScript. The experiment was successful enough that Macromedia approached Sun about licensing the Java 2 Micro Edition (J2ME) JVM for use in Flash. They wanted to use J2ME because the standard edition Java runtime was too large to embed within a Flash Web download. But Macromedia's proposed use of Java Micro Edition technologies did not align with Sun's Java licensing strategy. Edwin Smith, in a skunkworks effort, created a series of proof-of-concept virtual machines. Those VMs helped to convince Macromedia to build their own statically typed JVM-like virtual machine called AVM2 [Adobe 2007], and a new version of ActionScript to run on it. The new language was designed by Gary Grossman, Jeff Dyer, and Edwin Smith, and was heavily influenced by Horwat's draft $ES4_1$/JS2 specifications. However, like JScript.Net, ActionScript 3.0

[61] Descriptions of internal Macromedia beliefs and actions are derived from personal communications with Jeff Dyer and Gary Grossman, 2017–2018.

was a simplification of the $ES4_1$ design. It was less dynamic than JS2 and, unlike JScript.NET, it was not constrained by the .NET type model. ActionScript 3.0 was also similar to JScript.Net in that it was not heavily constrained by legacy compatibility concerns. Flash would ship with both AVM2 to support ActionScript 3.0 and AVM1 to support ActionScript 1.0 and 2.0. This effort to create a new version of ActionScript and a new virtual machine took over three years to complete. In was announced in 2006 as a component of Flash Player 9, which ultimately shipped in 2007. By the time the effort was completed, Adobe had acquired Macromedia and Flash had become Adobe Flash.

18 ES4, TAKE 2

Work on $ES4_1$ had stalled in 2003, but the use of JavaScript on the Web continued to grow. Within a year, TG1 members were again taking about designing a new version of ECMAScript that they called "ES4."

18.1 Resetting TC39-TG1

Macromedia became an Ecma member in November 2003, and Jeff Dyer became one of their TC39 delegates. This was an obvious move because the design of ActionScript 3 was heavily influenced by TG1's initial attempt to develop an ES4 specification. It was important to Macromedia to keep the design of ActionScript aligned with future ECMAScript specification work and that TG1 consider the requirements and precedents of ActionScript.

In the spring of 2004, the Mozilla Foundation had released a technology preview of the Firefox browser and was on track to make a Firefox 1.0 release before the end of the year. Brendan Eich, then Mozilla's CTO, had concerns about the future of the open Web. Interest in browser-hosted Web applications was growing rapidly, but the then-current suite of browser standards was inadequate to support rich applications. Closed proprietary application platforms such as Flash, Microsoft's Window's Presentation Framework (WPF), and .NET were competing to supplant the HTML/CSS/JavaScript Web technology suite. And the standards organizations responsible for the open Web were not responding to the challenge. In 1998, the W3C [W3C 1998] had decided to stop evolving HTML in favor of XML-based alternatives. However, XHTML was neither syntactically nor semantically compatible with HTML and was not universally accepted by browser vendors or Web developers. Similarly, Ecma TC39-TG1's attempt to evolve the ECMAScript specification had floundered and its attention had been diverted to designing ECMAScript support for XML processing. Some members of the Web technology community were concerned that "ECMAScript is Dead" [Schulze 2004b].

In response, Brendan Eich [2004] facilitated the formation of WHATWG—The Web Hypertext Application Technology Working Group [Hickson 2004]—which was focused on the future of HTML. He also started to reëngage with TG1. Eich met with the Ecma Secretary-General in March 2004 [Marcey 2004] and the Mozilla Foundation applied for Ecma membership in May. In June 2004, Eich attended a TG1 meeting [Schulze 2004a] for the first time since February 1998.

At the June meeting [Schulze 2004b], Convener responsibility for TG1 was transferred from Microsoft's Rok Yu to William Schulze of Macromedia. Jeff Dyer became editor of ECMA-262. The delegates rededicated themselves to completing a 4th edition of the ECMAScript specification but decided to not go forward with Waldemar Horwat's $ES4_1$ draft. According to Schulze's report, "[$ES4_1$ was] too sweeping and broad for completion or adoption." Instead, the members agreed to take "a more incremental approach" [Schulze 2004a] that could be integrated into existing implementations including ActionScript. Packages, namespaces, conditional attributes, runtime type checking and XML support were listed as candidate features for integration. This list included some of the most complex parts of the old $ES4_1$ draft, but members still committed to a 12-month

development cycle for a new Edition 4. Dyer agreed to prepare a draft of the contemplated changes for presentation at the meeting scheduled for October 2004.

TG1 was not able to meet these new commitments. Most of the committee's attention for the remainder of 2004 and much of 2005 remained focused on revising the E4X specification [Schneider et al. 2005] as part of an ISO fast-track process. Serious work on the new ES4 did not start until October 2005. However, during this interlude Brendan Eich familiarized himself with the then-current state of ECMAScript standardization and began to publicly express his ideas for the next edition in conference talks and blog posts [Eich 2005a,b]. At the September 2005 meeting [TC39-TG1 2005] Eich became TG1 Convener and started pushing to make progress on $ES4_2$ development.

18.2 Redesigning ES4

In an October 2005 blog post, Brendan Eich [2005d] enumerated four goals for the next round of work on ES4, as follows:

- Bringing Edition 4 back toward the current language, so that prototype based delegation is not a vestigial compatibility mode, but the dynamic part of an object system that includes classes with fixed members that cannot be overridden or shadowed.
- Allowing implementors to bootstrap the language,[62] expressing all the meta-object protocol magic used by the "native" objects (ECMA-262 Edition 3 section 15), including get/set/call/construct and control over property attributes such as enumerability.
- Adding type annotations without breaking interoperation of existing and new editions, in support of programming in the large which is needed more and more in XUL[63] and modern Web applications.
- Fixing longstanding problems that bite almost every JS hacker, as I've discussed previously.

He stated that his intention was to complete this work, including an initial implementation to test interoperability, by the end of 2006.

In a November 2005 blog post Brendan Eich [2005c] simplified these goals, as follows:

1. Support programming in the large with stronger types and naming.
2. Enable bootstrapping, *self-hosting*[g], and reflection.
3. Backward compatibility apart from a few simplifying changes.

He also stated that making ECMAScript more like Java or any other language and making ECMAScript more optimizable were non-goals. In subsequent presentations Eich [2006a] acknowledged criticisms of the original $ES4_1$ specification including those questioning whether declarative static types or class definitions were needed. He countered that doing nothing was not a viable alternative. His contention was that the ES3 language would scale poorly over the next ten years as Web developers built complex applications. In particular, he argued that a type system that could be used to enforce invariants and could optionally be statically checked was needed to enable such applications. But such a change could happen only once, so this was the time to do it.

Brendan Eich was optimistic that application of contemporary research in programming language specification techniques and type systems could help address some of the problem areas of the original $ES4_1$ work. In early 2006 he recruited Dave Herman to join the TG1 $ES4_2$ design team. Herman was a PhD student at Northeastern University and had worked on developing an operational semantics for ES3. Based upon Herman's recommendation, Eich also invited Cormac Flanagan, a

[62] Using JavaScript code to implement JavaScript's built-in library.

[63] XUL (XML User interface Language) was Mozilla's JavaScript framework for creating Firefox browser extensions.

Jeff Dyer	Adobe[65]
Brendan Eich	Mozilla
Cormac Flanagan	University of California, Santa Cruz
Lars T Hansen	Opera/Adobe
Dave Herman	Northeastern University
Graydon Hoare[66]	Mozilla
Edwin Smith	Adobe

Fig. 26. 2006 $ES4_2$ Core Design Team

professor at UC Santa Cruz, to join. Flanagan was an expert in hybrid type systems [Flanagan 2006]. At about the same time Lars Thomas Hansen, a software architect working on the Opera Web browser, became a regular TG1 participant.[64] Herman, Hansen, and Flanagan all had either direct or indirect ties to the programming language research community at Northeastern University.

In late 2005, TG1 established a schedule of weekly conference calls and monthly face-to-face meetings for the $ES4_2$ project. Figure 26 lists the core $ES4_2$ design team in 2006. These are the individuals who regularly attended the meetings, participated in key decisions, and made ongoing significant contributions. Other individuals from Adobe, Mozilla, and other organizations occasionally attended meetings and/or made contributions but were less actively involved in the project.

The first round of JS2/$ES4_1$ development had been quite cavalier about making changes which would be incompatible with existing ECMAScript programs. The assumption was that in browsers version information within HTML `<script>` elements could be used to select different versions of the language. The new $ES4_2$ effort was more cognizant of the potential impact of breaking-changes but still hoped to be able to use versioning to correct what the committee considered to be early JavaScript design errors. Brendan Eich spoke about this possibility in his blog posts and presentations. But there was also push back from some TG1 members. Douglas Crockford, representing Yahoo! at the July 2006 TG1 meeting [TC39-TG1 2006c], stated that "backwards compatibility was hard & important" but security was their biggest problem and that backward incompatibilities could be tolerated if they fixed security related issues. Pratap Lakshman of Microsoft stated: "Priority 0 [the highest priority] is backwards compatibility. *Only* for security fixes will backwards compatibility be broken."

Brendan Eich had said positive things about Python during the Q&A session [Danvy 2005] after his ICFP'05 keynote talk commemorating the tenth anniversary of JavaScript. He even speculated that for larger Web scripts, Python could have been a better language than JavaScript. Over the course of the next year he lobbied for the inclusion of several features in $ES4_2$ that were directly modeled after equivalent Python features. These included iterators, generators, *destructuring*[g] assignment, and array comprehensions. He also promoted the concept of block-scoped variables declared using the `let` and `const` keywords as an alternative to function-scoped `var` declarations. These features were largely orthogonal to the other more complex "programming in the large" features (as they were called) proposed for $ES4_2$ and versions of them were added to the SpiderMonkey-based JavaScript 1.7 engine [Mozilla 2006a], shipped as part of the Firefox 2 browser in October 2006. However, those features were not adopted by other browsers and hence did not have significant use outside of XUL.

[64] Effective April 2007, Hansen represented Adobe.
[65] Adobe completed its acquisition of Macromedia on December 3, 2005.
[66] In 2006 Hoare, as a personal project, was working on the early design of the Rust [Hoare 2010] programming language.

Eich was concerned that other browser vendors and in particular Microsoft would be slow to adopt ES4$_2$'s JavaScript improvements. Also of concern was the possibility that JavaScript engines would not improve their performance to meet the demands of the rapid emergence of AJAX Web applications. One way to address both issues would be to make a high-performance open-source JavaScript engine available that supported the anticipated ES4$_2$ specification. To this end, Eich convinced Adobe to contribute their AVM2 implementation to Mozilla under an open-source license. Mozilla named the resulting code base "Tamarin" [Mozilla 2006b]. In subsequent months, Mozilla announced [Eich 2007a] two projects: ActionMonkey, whose goal was to use the Tamarin code base as a replacement for SpiderMonkey; and, ScreamingMonkey, a Tamarin-based JavaScript engine that could be added as a third party plug-in extension to Internet Explorer. Neither project was completed.

While this industry maneuvering was taking place, TG1 continued working on the new ES4$_2$ design. A major goal of ES4$_2$ was to provide a type system and type annotation notation that could be used to validate the usage of complex data abstractions in large programs. Static type analysis prior to deployment should be possible for suitably written programs, but the type system needed to be able to deal with both new and existing unannotated programs and with the dynamic structural mutation of objects allowed in the existing language. Much of the committee's time in 2006 was devoted to understanding the implications of these requirements and trying to design a type system to accommodate them [TC39-TG1 2006a,d].

The committee started with the type system informally described in the ActionScript 3 specification [Macromedia 2005]. This was a nominal type system with class and interface types similar to Java prior to the addition of generic types. ActionScript 3 has type-annotated declarations and includes a universal type for declarations that lack an explicit type annotation. The ActionScript 3 specification does not explicitly include the concept of function subtypes and has an incomplete definition of class/interface subtyping. The language has a strict mode that performs ahead-of-time static type-checking using type-annotated declarations and limited type inferencing, and a standard mode that dynamically validated actual data values against type annotations and operational requirements.

An early suggestion of Dave Herman and Cormac Flanagan was to use a contract model [Findler and Felleisen 2002] to better unify strict and standard modes along with typed and untyped declarations. As work progressed, structural types [TC39 ES4 2006d] were added to deal with object and array literals and parameterized types were added to deal with array types. Many alternatives [TC39 ES4 2006b] were considered and documented on TG1's private[67] wiki site [TC39 ES4 2007g]. Herman and Flanagan also experimented with formalization of the type system [TC39 ES4 2007a]. By early 2007 the design was still incomplete but had evolved to encompass many modern typing concepts including function types and co/contra-variance considerations [TC39 ES4 2007b]. The realities of supporting optional typing and legacy dynamically typed programs was an ongoing significant source of complications.

Throughout 2006 and most of 2007 TG1 continued to work on developing new proposals and refining existing proposals. Eventually there was a list [TC39 ES4 2007e; Appendix L] on the private wiki of fifty-four approved proposals slated for inclusion in the ES4$_2$ specification. An additional twenty-six proposals [TC39 ES4 2007f] had been deferred or dropped.

Dave Herman had been recruited to TG1 after Brendan Eich discovered Web pages [Herman 2005] documenting Herman's experiments in formalizing the semantics of the ES3 specification. At the February 2006 TG1 meeting [TC39-TG1 2006b] Herman presented an introduction to formal techniques for specifying programming languages. He explained that in addition to providing

[67] TC39's private wiki was eventually made public as wiki.ecmascript.org [TC39 2007].

guidance to implementors, a formal specification provides a way to find and correct bugs within the specification. Concerns were raised about whether such a formal specification would be readable by ECMAScript implementors and other users of the specification. Herman felt that an operational-semantics-based formalism could be made quite readable. Over the next few months Herman explored using Maude [Clavel et al. 2003], Stratego [Visser 2001], and PLT Redex [Matthews et al. 2004] for specifying the ECMAScript semantics but ultimately found them unsatisfactory for the task. Over the same period there were also discussions about the possibility of defining the language in terms of a reference implementation. Another possibility was to design a new formal specification language specifically for ECMAScript. In October 2006, there was discussion [TC39-TG1 2006e] of the possible syntax and semantics of such a language until Cormac Flanagan pointed out that the committee was now talking about taking on the work of defining two languages, the specification language and the new version of ECMAScript. At this point the group quickly agreed to use an existing language to write a definitional interpreter for $ES4_2$. they quckly decided on using the language SML[68] [Milner et al. 1997]. By the middle of November, TG1 had put in place the tools and infrastructure for this effort and members were working on coding the interpreter. Herman and Flanagan [2007] describe the impact this had on the working style of the committee, as follows:

> Once we switched to a definitional interpreter, the interaction style of the committee changed substantially, from monthly 1½-day discussion-oriented meetings to a 3-day *hackathon*[g], interspersed with technical discussions, as various corner cases in the language design and implementation were discovered and resolved.

18.3 Resistance

Microsoft had minimal involvement with the restarted $ES4_2$ effort. Microsoft's Developer Division (DevDiv) had always been responsible for JScript development even though DevDiv was organizationally remote from the Microsoft Windows organization which was responsible for Internet Explorer. In the early 2000's DevDiv had reorganized in support of the .NET initiative and its C# product unit was given responsibility for both JScript.NET and the more conventional JScript engine used within Internet Explorer. This included the responsibility for participating in ECMAScript standardization activities. However, with weak customer adoption of JScript.NET and with the Windows organization having little interest in enhancing Internet Explorer, JScript/ECMAScript work was a low priority activity within the C# group.

In the 2000s, Microsoft usually sited strategically important development efforts at its main Redmond, Washington, campus and often sited more tactical projects at other campuses around the world. For the 2006 fiscal year, July 2005–June 2006, Microsoft DevDiv decided to transfer responsibility for all JScript/ECMAScript work to its India Development Center (IDC) in Hyderabad. DevDiv had previously transferred responsibility for its Java-like J#.NET product to IDC [Prasanna 2002]. By the spring of 2006 the transfer was largely complete. The task of representing Microsoft at TG1 was given to Pratap Lakshman who had worked on the J# team and had also been involved with TC39-TG3, the Ecma C# standards task group. Lakshman remotely attended his first TG1 meeting in April 2006 and started attending phone meetings and some face-to-face meetings. However, he was not a significant contributor to the $ES4_2$ development work during that period.

Allen Wirfs-Brock, one of the authors of this paper, had joined Microsoft in 2003 as a software architect on an exploratory project investigating new IDE architectures. Prior to Microsoft, he had been deeply involved for two decades with the Smalltalk programming language and development environments. Wirfs-Brock had been the lead developer of one of the first commercial Smalltalk

[68] Standard ML of New Jersey.

virtual machine implementations [Caudill and Wirfs-Brock 1986], worked on Smalltalk enhancements to support programming in the large, designed the standard Smalltalk exception handling system, and wrote the language definition portion of the ANSI Smalltalk Standard [ANSI X3J20 1998].

By late 2006, the IDE project seemed to be running its course and Wirfs-Brock had his eyes open for new opportunities. At this time there was growing interest in dynamic languages within DevDiv. Because no single DevDiv product unit was then currently responsible for dynamic languages; the various product unit managers were jockeying to do something with them. Wirfs-Brock took a staff architect position reporting to Julia Liuson, the Visual Basic Product Unit Manager, to advise her on dynamic language technologies and opportunities.

Allen Wirfs-Brock started his new position the first week of January 2007. During a casual conversation, Liuson asked if he knew anything about JavaScript. Wirfs-Brock recalls responding with something like: not much, it's a dynamic language used on Web pages that I think is loosely related to Self. Liuson then turned her monitor around and showed him an email message she had just received and asked whether he had any thoughts about it.

The message was from Pratap Lakshman and addressed to all DevDiv Product Unit Managers. It asked for guidance on the position he should be taking regarding a new JavaScript standard that Ecma TC39 was developing. Wirfs-Brock's recollection is that Lakshman's message said the new standard was based upon Adobe Flash and that it was going to be a substantial change from what was then currently in browsers. Lakshman said that what TC39 was developing was a powerful language that was likely to be too complicated for the Web. He went on to enumerate a long list of new features and changes including class-based static typing, structural types, parameterized types, and method overloading. He also stated that the revised language would be specified via a reference implementation written in Standard ML.

Allen Wirfs-Brock's response to Julia Liuson was that this sounded like a complete redesign and that in his experience attempts to improve dynamic languages by adding static types were seldom successful. He did not know enough about JavaScript or Web development to say anything more definitive. However, he offered to research it further.

Wirfs-Brock spent several days familiarizing himself with JavaScript, the then-current ES3 specification, and TG1 proposals from the public wiki snapshot [TC39 ES4 2007g]. He spoke with Lakshman, software architects on the Internet Explorer team, and Microsoft engineers working on Web-based applications. He recognized JavaScript's role on the Web as being a significant instance of Richard Gabriel's [1990] "Worse Is Better" concept. It was a minimalist creation that had grown in a piecemeal manner to become deeply ingrained in the fabric of the World Wide Web. In contrast, the $ES4_2$ effort appeared to Wirfs-Brock to be what Gabriel calls a "do the Right Thing" project that was unlikely to reach fruition and, if it did, would be highly disruptive to the Web. He concluded that the technically responsible thing to do would be to try to get ECMAScript evolution back onto a path of incremental evolution.

Given Microsoft's then lack of strategic interest in Web browser technologies, Wirfs-Brock thought it unlikely that DevDiv management would be interested in allotting resources to a Web browser related effort. He decided that, for internal DevDiv consumption, he would need to focus on the possible consequences of $ES4_2$ becoming a success. The primary concern that he identified was Adobe's contributions of the ActionScript 3 language definition and virtual machine to the effort. DevDiv's proprietary focus was on its .NET platform and its flagship language, C#, whose primary customers were enterprise application developers. The main competition for .NET was Sun's Java platform but DevDiv was starting to see Adobe's ActionScript-based Flash and Flex products as .NET competition. Wirfs-Brock anticipated that a successful $ES4_2$ effort could transform ActionScript into a first-tier enterprise language comparable in power and utility to C# or Java.

That, in combination with its standardization as the primary language for Web development, could result in a serious competitive threat to Microsoft's languages and developer products.

Allen Wirfs-Brock wrote a memo stating these concerns and recommending that Microsoft actively engage within TG1 and try to redirect it onto a path of piecewise, non-disruptive evolution of the ECMAScript standard. By mid-January, that recommendation was accepted and Wirfs-Brock was given the responsibility of carrying it out. On January 18, 2007, Pratap Lakshman posted a message to the TG1 private mailing list [TC39 2003] introducing Wirfs-Brock as a new Microsoft TG1 delegate.

The March TG1 face-to-face meeting was to be hosted by Microsoft and Wirfs-Brock decided to make that the first meeting he would attend. But he also felt that it was important to quickly disabuse the committee of its perception that Microsoft was supportive of the $ES4_2$ effort. He asked Pratap Lakshman to convey that message at the February meeting. Lakshman did so and also posted [Lakshman 2007a] a page to the TG1 private wiki where he floated the idea of a simplified ES4 browser profile. He reported back that the response he received was quite hostile but that during a coffee break he was approached by Douglas Crockford who offered that Yahoo! would be willing to stand with Microsoft in opposition to $ES4_2$.

Allen Wirfs-Brock contacted Douglas Crockford and they agreed to work together to create a joint Microsoft-Yahoo! proposal for an alternative to the $ES4_2$ project. Crockford [2002d] had previously published a small set of recommended modifications to the ECMAScript language that were intended to make the language "a little bit better" by correcting mistakes and inconveniences in its original design. Wirfs-Brock and Crockford agreed that they would use those recommendations as the starting point for the technical aspect of a joint proposal. Pratap Lakshman, as a follow-up to his browser profile idea, posted a proposal [Lakshman 2007b] for a minimalist approach that incorporated many of Crockford's suggested ES3 modifications. Meanwhile, Wirfs-Brock collaborated with Crockford and Lakshman to draft a more formal proposal that was circulated within both Microsoft and Yahoo! for internal approvals. On March 15, 2007, ahead of the March 21–23 TG1 meeting they posted the proposal [Crockford et al. 2007] and Crockford announced it via the TG1 private email distribution list.

The full title was "Proposal to Refocus TC39-TG1 On the Maintenance of the ECMAScript 3rd Edition Specification" and its opening paragraph was as follows:

> We believe that the specification currently under development by TC39-TG1 as ECMA-Script 4 is such a radical departure from the current standard that it is essentially a new language. It is as different from ECMAScript 3rd Edition as C++ is from C. Such a drastic change is not appropriate for a revision of a widely used standardized language and cannot be justified in light of the current broad adoption of ECMAScript 3rd Edition for AJAX-style Web applications. We do not believe that consensus can be reached within TC39-TG1 based upon its current language design work. However, we do believe that an alternative way forward can be found and submit this proposal as a possible path to resolution.

The proposal recommended that TG1 should be reconstituted around three work items. The first work item was the maintenance of the then-current ECMAScript language defined by the 3rd Edition specification. The maintenance work would include clarification of underspecified portions of the 3rd Edition; incorporation of new features such as those in Mozilla's JavaScript 1.6/1.7; and, minor corrections and improvements such as those identified by Crockford. The second work item was to draft a standard definition for ActionScript. The third work item was to define a new programming language for the browser that could coexist with ECMAScript while not being constrained by ECMAScript compatibility. The proposal left open the possibility that work items

two and three might be merged. It suggested that work items two and three be assigned to a new TC39 Task Group distinct from TG1.

As expected, the response on the TG1 private mailing list[69] was generally negative, but it did reveal that Apple's Maciej Stachowiak [2007b] also had reservations about the direction $ES4_2$ was taking. Brendan Eich [2007b] was the most vocal respondent defending static typing and other $ES4_2$ features as being essential to improved performance and the structuring of large applications. He also questioned both Microsoft's and Yahoo!'s motivations for making the proposal [Eich 2007c].

The email discussion intensified as the March meeting date approached. Pratap Lakshman requested that most of the second day of the meeting be devoted to the Microsoft/Yahoo! proposal. Brendan Eich countered that an hour should be sufficient, and both he and Jeff Dyer expressed the desire that the majority of the meeting continue as an $ES4_2$ hackathon. Both Eich and Dyer argued that the $ES4_2$ activities represented the long established TG1 consensus that Microsoft had helped form and questioned whether it was appropriate for Microsoft and Yahoo! to now try to break that consensus. Allen Wirfs-Brock responded that consensus was already broken because Microsoft and Yahoo! were two of the three Ordinary Members of Ecma who regularly participate in TG1.

The second day of the March meeting [TC39-TG1 2007c] was more heavily attended than usual. In addition to Allen Wirfs-Brock and Pratap Lakshman, Microsoft was represented by Scott Isaacs and Chris Wilson. Isaacs was a framework architect for Microsoft's "live.com" Web applications and had been one of the original developers of DHTML.[70] Wilson was the platform architect for Internet Explorer and was actively involved with W3C Web standards. Isaacs and Douglas Crockford both spoke about the difficulties in Web application development when there is poor interoperability among ECMAScript implementations within browsers. Crockford argued that a more complete specification of ES3 level functionality would improve the stability of the Web by helping to eliminate interoperability issues. Isaacs was particularly concerned about minimizing new syntactic language extensions that could cause parsing errors for new Web pages in older browsers. Both Isaacs and Crockford emphasized the growing importance of security and privacy features within Web applications. Eich, Dyer, and Graydon Hoare countered that $ES4_2$'s type system was the foundation needed for a more stable, secure, and performant browser programming environment. Wirfs-Brock argued that an evolutionary "ES3.1" specification would help stabilize the Web and provide time for ES4 to be implemented and adopted. Eich was concerned that this was simply a delaying tactic to give Microsoft time to establish their .NET-based Rich Internet Application Web platform[71] as competition for the standards-based HTML/CSS/JavaScript platform. He cautioned that there was already a lot of community buzz and excitement about ES4 and that it would reflect negatively on Microsoft and Yahoo! if they forced a delay in its development.

Ultimately there was agreement that there might be some value in developing an "ES3.1" specification and that Microsoft and Yahoo! could work on it within the context of TG1. This was the outcome that Wirfs-Brock had hoped for when preparing for the meeting. The $ES4_2$ proponents insisted that ES3.1 must be a subset of $ES4_2$ and that its specification must use the specification style developed for $ES4_2$. Wirfs-Brock was not particularly concerned about those limitations as he still believed that it was unlikely that an $ES4_2$ specification would ever be completed and released.

Pratap Lakshman, Allen Wirfs-Brock, and Douglas Crockford started working on defining the ES3.1 project. Wirfs-Brock and Crockford met on March 29 and agreed that Lakshman should draft an initial proposal that could be circulated prior to the April TG1 meeting. Crockford suggested some design principles and that the 3.1 specification be in the same style as the ES3 specification,

[69] Ecma International stores an archive of this mailing list [TC39 2003]. What follows is based upon a review of that archive.
[70] Dynamic HTML.
[71] This platform, initially code-named WPF/E, was still in its prerelease preview phase at this time. It was released in April 2007 with the product name "Silverlight."

Goals

1. Improve implementation conformance by rewriting the specification to improve its rigor and clarity, and by correcting known points of ambiguity or under specification.
2. Add commonly implemented and used extensions to the standard (specifically most JavaScript 1.6 and 1.7 features)
3. Incorporate high leverage incremental extensions that support current usage experience and best practices
4. Adopt low impact language changes that correct well known performance or reliability issues
5. Identify problematic features to be designated as deprecated
6. Maximize both forward and backward compatibility between ES3 and ES3.1 as well as between ES3.1 and ES4.

Design Principles

1. Primary focus is on correction of known errors and clarification of known ambiguities.
2. New features only considered if:
   a. They introduce no new syntax
   b. Offer significant new value
3. Prefer features that have been proven in existing implementations
4. Features may be marked as deprecated if they are known to create significant security or reliability issues.
   a. Consider deprecating features with minimal value that cause significant performance related issues.

Fig. 27. ES3.1 Initial Goals and Design Principles [Lakshman et al. 2007]

even though that would conflict with the agreement at the March meeting. Using the same specification formalisms was problematic when the final form of the $ES4_2$ specification had not yet been established.

On April 15 Pratap Lakshman posted a number of pages to the wiki under the title "ES3.1 Proposal Working Draft" [Lakshman et al. 2007]. It included a set of goals, backward/forward compatibility requirements, and design principles (Figure 27). It also included descriptions of approximately twenty fixes, changes, and new features that were candidates for inclusion. Many of these were derived from Douglas Crockford's "Recommended ECMAScript Changes" document which he had updated in early April and would update twice more as he contributed to ES3.1 [Crockford 2007b,c,d].

The ES3.1 working draft was discussed at the April meeting [TC39-TG1 2007a]. The major concern of the $ES4_2$ developers was how the ES3.1 work would relate to the $ES4_2$ specification. They wanted the ES3.1 work to follow the same ML reference implementation specification technique that they intended to use for $ES4_2$. The ES3.1 group pushed back that it did not seem very useful to completely change the specification technique for a maintenance release of a specification. Jeff Dyer finally suggested that, given the difference of perspective, ES3.1 people should just continue with what they were doing. But he warned that work done in the context of the ES3 specification would be of little interest to the rest of the group.

Through the rest of the spring and summer of 2007, the two subgroups largely worked independently on their two projects. The ES3.1 group was analyzing the existing ES3 specification and its implementations to identify interoperability issues that existed because of underspecification or from failure to follow the specification [Lakshman 2007c; Wirfs-Brock 2007b; Wirfs-Brock and

Crockford 2007]. The $ES4_2$ group continued to use its ML reference implementation as a tool to flesh out their various proposals.

The $ES4_2$ project continued to be very aggressive with its schedule. At the beginning of May 2007, a report [Miller 2007] to the Ecma Co-Ordinating Committee stated that the final draft of the $ES4_2$ specification would be finished by October 2007 so that the Ecma General Assembly could approve it in December. On June 8, 2007, Dave Herman [2007; Appendix K] announced on *Lambda the Ultimate*[72] the availability of the "M0" release[73] of the ES4 Reference Implementation [TC39 ES4 2007d].

At the June meeting [TC39-TG1 2007b] there was a call-to-action to immediately start the ES4 "spec writing process." But, significant technical design issues remained unresolved and new issues were being frequently discovered. For example, at the July meeting [Eich 2007d] it was recognized that there were significant issues with performing run-time type-checking of structural types.

The September 7 TG1 Convener's Report [Eich 2007d] stated that a 2007 completion date was unrealistic and that the new completion date was being pushed out a year to September 2008. It also reported that Lars Hansen was taking on the role of $ES4_2$ editor. The report did not mention the ongoing ES3.1 work or the reservations of Yahoo! and Microsoft about $ES4_2$.

A goal of the September meeting [TC39-TG1 2007d] was to accept, reject, or defer for a future edition all remaining unresolved $ES4_2$ proposals that existed on the ES4 wiki. From the perspective of the $ES4_2$ working group, that included the proposal labeled "Maintenance of ES3" which was the umbrella proposal for ES3.1 work. Jeff Dyer's position at the meeting was that before the end of the day this proposal needed to be either accepted or rejected (and marked as such on the wiki). If rejected, that would remove it as a work item for TG1. It is apparent in the meeting minutes that he did not believe that acceptance was a possible outcome. Brendan Eich's position was more nuanced. As the public champion of $ES4_2$ he found the ES3.1 effort a distraction and was very skeptical of Microsoft's motives. He did not want ES3.1 to be developed in competition with $ES4_2$ and suggested that the ES3.1 proponents consider leaving TG1 and seeing whether TC39 would be willing to establish a new TG to accommodate them. However, as the TG1 convener, he wanted to find a way to avoid splitting the group. He suggested that the work product of the ES3.1 group might be published as an Ecma Technical Report or some other less formal, non-ISO, non-standards-track document. The entire conversation was heated and stressful for both the $ES4_2$ and ES3.1 proponents. At one point Pratap Lakshman in frustration stated, "We do not support or agree to the current ES4 proposal, either in whole or in part. We intend to continue to work with interested TG members to develop a proposal for a more incremental revision of the current specification." This reflected Microsoft's position though it was not a very politic statement and was slightly inaccurate regarding all parts of $ES4_2$. Ultimately, the immediate problem of the status of the "Maintenance of ES3" proposal was resolved by moving the ES3.1 pages from under the "Proposals" namespace of the wiki and placing them under a new "ES3.1" namespace. However, the conflicting goals of the ES3.1 and $ES4_2$ proponents were not resolved and soon spilled out into public rhetoric [Kanaracus 2007].

18.4 Finding Harmony

During 2007, the set of active TG1 participants started to grow. Some of the growth was due to efforts by both the ES3.1 and $ES4_2$ groups to encourage new and currently inactive members to participate. In the spring, inactive TG1 members IBM and Apple began to more regularly send representatives to TG1 meetings and participate in online discussions. Google joined Ecma as

[72] A weblog that was popular among programming language researchers and implementors. Also known as LtU.
[73] M0 is an abbreviation for Milestone 0.

an Ordinary Member and appointed Waldemar Horwat as its GA representative and lead TG1 delegate. The Dojo Foundation joined as a non-profit member represented by Alex Russell and Chris Zyp. Both Allen Wirfs-Brock and Douglas Crockford encouraged Mark S. Miller, an expert in object-capability (OCAP) languages[74] [Miller 2006], to participate. Miller worked for Google and he started attending meetings as a Google delegate. Some of the new participants brought a Web developer's perspective to the group which previously had been dominated by language designers and engine implementors.

At the beginning of 2007, TG1's goal was to have a finished $ES4_2$ specification by October. That goal was not met, but in October Lars Hansen [2007e] completed a document whose initial drafts [Hansen 2007b] were titled "ECMAScript 4th Edition Language Overview." This was not a detailed specification but instead a 40 page summary of the major features of the language. The first paragraph of its abstract described the language in this way:

> The fourth edition of the ECMAScript language (ES4) represents a significant evolution of the third edition language (ES3), which Ecma approved as the standard ECMA-262 in 1999. ES4 is compatible with ES3 and adds important facilities for programming in the large (classes, interfaces, namespaces, packages, program units, optional type annotations, and optional static type checking and verification), evolutionary programming and scripting (structural types, duck typing, type definitions, and multi-methods), data structure construction (parameterized types, getters and setters, and metalevel methods), control abstraction (proper tail calls, iterators, and generators), and introspection (type metaobjects and stack marks).

It ultimately proved to be the best overall description of the envisioned $ES4_2$ language. However, both Allen Wirfs-Brock [2007c] and Douglas Crockford [2007a] expressed concern that the unqualified use of the name "ECMAScript 4th Edition" suggested that it was describing a language that was very close to final approval as an Ecma standard. In addition, the introduction to the document presented the design as representing the consensus of Ecma TC39-TG1 and made no mention of any dissenting opinions about the $ES4_2$ design within TG1. After some negotiations, Hansen agreed to add "Proposed" as the first word of the title and to insert a paragraph into the document's introduction which stated that a minority of TG1 opposed standardization of the presented design. Similar concerns were raised about the website [TC39 ES4 2007c] that was being set up by members of the $ES4_2$ team to distribute the overview paper and the Reference Implementation code. These incidents intensified the ES3.1 proponents' concerns about how the $ES4_2$ proponents were publicly marketing ES4 while continuing to ignore or discount the ES3.1 effort.

Allen Wirfs-Brock was in regular contact with Microsoft's corporate standards group including, Isabelle Valet-Harper who was a member of the Ecma Co-Ordinating Committee (CC). The CC [Ecma International 2007b] was concerned that the private, externally hosted wiki TG1 used for documents and meeting minutes was not accessible to the Ecma secretariat and general membership. The Secretariat requested that copies of agendas, meeting notes, and important documents be formatted for posting to Ecma's internal members-only website. TG1 decided that the easiest way to comply was to make the entire TG1 wiki publicly readable [TC39 2007].

At the October 2007 CC meeting [Ecma International 2007a] there were discussions about the operation of TC39-TG1. Prior to 2001, TC39's charter had involved only ECMAScript. In 2001, it had expanded to encompass additional programming languages and platforms, each the responsibility of a largely independent TG. ECMAScript development was assigned to TC39-TG1. The Ecma Secretariat was generally focused on supervising and supporting TC-level activities rather than those of TGs. By 2007, TG1 seemed to be operating autonomously without supervision from either

[74] Using objects as the foundation of a capability-based access control system.

TC39 or the Secretariat. Some CC members were concerned that TG1 might not be following all of Ecma's policies and procedures. Also discussed was the reported lack of consensus within TG1 on its then-current work. One possible solution discussed was to elevate TC39-TG1 to full TC status so it would receive greater Secretariat oversight. John Neumann, the then-current Ecma President, agreed to attend the November 2007 TG1 meeting to try to clarify the situation.

That meeting [TC39 2007] was devoted mostly to airing the CC's concerns and discussion of the apparent lack of consensus about the ES3.1 and $ES4_2$ projects. John Neumann emphasized concerns about the lack of communications of venue notices, agendas, meeting minutes, and key documents from TG1 to the rest of Ecma and insisted that this needed to change. He also cautioned that, from an Ecma perspective, TG1 was being too publicly open in some cases. In particular, there was concern within Ecma's management that disagreements among TG1 members were being argued in public on weblogs and discussion forums. Neumann announced that he was going to recommend that ECMAScript-related activities again become the single focus of TC39. Essentially, TC39-TG1 would be rechristened as TC39. This would make the ECMAScript work more visible within Ecma and make the support and oversight of the Ecma secretariat directly available to it. The other currently active TGs of TC39 would be transferred into a a newly created TG49. This reorganization was approved by the Ecma General Assembly at its December 2007 meeting, and as of January 2008 TC39-TG1 once again became just TC39.

The November meeting also included discussion about TC39's charter going forward. Douglas Crockford proposed that there should be a new project to define a *Secure ECMAScript*[g] (SES) that could support *mashups*[g] and other security-sensitive applications. Allen Wirfs-Brock [2007] distributed a new Microsoft Position Statement that reiterated its call to take an evolutionary approach to moving the ES3 language and specification forward rather than continuing with the existing $ES4_2$ effort. Crockford announced Yahoo!'s support of that position. Lars Hansen asserted that "the 3.1 proposal has languished and was finally sidelined in September, we're working on ES4 here, not 3.1." Brendan Eich also claimed that not much had happened with ES3.1 since April. Wirfs-Brock did not accept that ES3.1 was sidelined and pointed out that multiple documents [Lakshman 2007c; Wirfs-Brock 2007a,b; Wirfs-Brock and Crockford 2007] analyzing ES3 interoperability issues had been produced as inputs into ES3.1 development.

A straw poll was taken to gauge interest in the three possible TC39 activities. Everybody present (representing nine organizations) supported continued work on $ES4_2$. Continuing work on ES3.1 was supported by Microsoft, Yahoo!, Apple, Google, and Mozilla. Starting work on a Secure ECMAScript was supported by Microsoft, Yahoo!, Apple, and Google. From an Ecma perspective, those levels of support were more than adequate to justify moving forward with those activities within the new TC39. Microsoft's support for continuing $ES4_2$ was contrary to what was stated in its position paper. Allen Wirfs-Brock recalls thinking that it was unnecessary to push further on that point because he still expected the $ES4_2$ effort to ultimately fail.

After the December 2007 Ecma GA meeting, Isabelle Valet-Harper spoke with Allen Wirfs-Brock about who might be an appropriate chairperson for the new TC39. Brendan Eich could not be chair because Ecma's then-current rules required that TC chairs represent an Ordinary Member. Mozilla was a Not-For-Profit Member. Wirfs-Brock and Valet-Harper agreed that the ideal chair would be somebody who did not have a personal investment in or opinion regarding ES4, ES3.1, or any other possible TC39 project. Valet-Harper suggested that Microsoft and Adobe, in a spirit of cooperation, each contract with John Neumann to represent them and jointly nominate him as TC39 chair. Adobe agreed to this idea and it was announced at the January 2008 TC39 meeting and at the March 2008 meeting Neumann was elected as the TC39 chairperson.

In November, 2007, Lars Hansen [2007c] prepared an "Editor's Report" with a new schedule aiming for a final $ES4_2$ draft by October 2008 and publication as an Ecma standard in December 2008.

He also wrote a paper [Hansen 2007a] summarizing intentional $ES4_2$ incompatibilities with ES3, and a tutorial [Hansen 2007d] on how $ES4_2$ gradual typing supported evolutionary programming. In February 2008, Jeff Dyer [2008a] posted a new work plan, still targeting December, with intermediate drafts in May, July, and September. Hansen and Dyer [2008] also posted a position statement titled "Features to Defer From Proposed ECMAScript 4." It argues that the then-current $ES4_2$ plan includes a number of features which are "strange, unproven, or costly" and that deferring them

> will significantly increase the likelihood of finishing the spec in 2008, will increase community buy-in, will help keep implementation complexity manageable, will reduce the risk of standardizing something we'll later regret, and will—with plenty of luck—somewhat reduce the discord among TG1 members.

Proposed deferrals were: numeric conversion, int and uint, decimal, operator overloading, generic functions, `wrap`[75], stack marks, generators, tail calls, nullability, program units, reformed `with`, resurrected `eval`, and namespace filters. After justifying why each of those features should be deferred, the position statement presents Adobe's revised view on how ECMAScript should evolve going forward:

> We think ES needs to evolve in a more piecemeal fashion than we're seeing for ES4 so far. The fact that nine years will have passed from the publication of E262-3 to the publication of E262-4 is not in itself a valid reason to introduce a large number of new features at once; each feature must carry its weight, and experience must guide us. That said, this paper [is] not advocating a watered-down "ES3.1" (which should really be called "ES3.01"); we are advocating that we go for the 80% solution "ES3.8" now and then plan to grow to meet new needs in the near future, when those needs are clearer.

There is no substantive discussion of this position paper recorded in any of the TC39 meeting minutes or in the private or public TC39 email channels. The only recorded response was IBM objecting to the suggestion that decimal arithmetic should be excluded. During this same period significant criticism of various aspects of the $ES4_2$ design, methodology, and process was posted to the *es4-discuss*[g] mailing list. Some criticism came from influential framework developers and ECMAScript implementors at Apple and Google. In March 2008 the $ES4_2$ designers discovered [Dyer 2008b] that there were fundamental semantic issues with the $ES4_2$ package abstraction used to define modules and in May issues with namespaces were identified [Stachowiak 2008b].

Throughout the spring of 2008, Lars Hansen posted initial drafts of individual $ES4_2$ specification sections for feedback. On May 16, Hansen [2008] announced his first draft [Hansen et al. 2008a,b,c] for the specification:

> Enclosed is a quite incomplete first draft of the specification for the Proposed ECMA-Script 4th Edition. This draft is comprised of a short introduction, the surface grammar, and a description of the core semantics—values, storage, types, names, scopes, and name resolution. More will follow as it is ready, probably on a (more or less) bimonthly schedule.

During this same period, the ES3.1 subgroup started developing a specification derived from the ES3 specification. There was an expanding set of participants from organizations such as Google, IBM, Dojo Foundation, and Apple. The initial draft of the ES3.1 specification was distributed [Lakshman 2008; Lakshman et al. 2008] on May 28.

[75] In $ES4_2$, `wrap` is an operator that performs a dynamic structural type check on a value and then, if the type check succeeds, creates a wrapper object which can be used in place of that value. The wrapper revalidates each operation applied to it before delegating the operation to the original value. Wrappers allow objects, whose properties may be removed or modified, to be used in the context of statically provided type declarations.

At the May 29–30, 2008, meeting, both specifications were introduced by their editors. Detailed discussion was deferred until the July meeting to give members time to read the specifications. It was clear from the rate of progress, the amount of remaining writing, and the number of still unresolved design issues, that the final $ES4_2$ specification could not be ready for December 2008. June 2009 or more likely December 2009, should be the release target. For ES3.1 to be ready for December 2008, all major design decisions needed to be finalized before the July 2008 meeting. June 2009 seemed like a more realistic target.

In late June 2008, John Neumann organized a conference call that included Brendan Eich, Allen Wirfs-Brock, Douglas Crockford, Adobe's Dan Smith[76], and David McAllister, Adobe's Ecma General Assembly representative. McAllister and Smith announced that Adobe was going to discontinue its support of the $ES4_2$ effort and that the staff assigned to it was going to be moving on to other activities. Everybody present understood that this was the end of $ES4_2$ and that a broader announcement should be carefully orchestrated. They agreed to make the announcement to all of TC39 at its upcoming July meeting and to decide at that meeting how to make a public announcement. Eich, who had been forewarned of the decision by Adobe, was in agreement with it and expressed a hope to harmonize all of TC39 around completing the ES3.1 effort and developing a common plan for the future that was not constrained by past ES4 design decisions. He agreed to present that vision at the upcoming meeting. The agenda for the July 23–25 meeting in Oslo, Norway, was revised [TC39 2008f] to list "Harmonization of ECMAScript" as the first item of new business.

In 2018 email discussions, Jeff Dyer and Lars Hansen recounted that the withdrawal was their decision made in consultation with their manager, Dan Smith. They had become convinced that $ES4_2$ was unlikely to get finished. Their perception was that opposition from the members of the ES3.1 working group was stalling work on $ES4_2$ and it was becoming apparent that the status quo approach of ES3 plus fixes was carrying the day within TC39, leaving no room to incorporate the static features of ActionScript 3.

Cormac Flanagan, in a 2019 personal communication, speculates that Adobe's withdrawal was really a recognition of the problems with $ES4_2$. His postmortem thoughts also include the following:

- The substantial language extension planned for ES4 was (in retrospect) a high-risk, non-conservative approach.
- There was cutting edge language [technology] involved in the standardization process, particularly around the addition of the static type system (10+ years later [in 2019], there are still hard unsolved research and performance problems [Greenman et al. 2019]). The publication of "Space-Efficient Gradual Typing" at TFP'07 [Herman et al. 2011], inspired by performance concerns in ES4, is perhaps a reflection of the researchy nature of this work.
- The 'buy-in' concerns around ES4 in TC39, while problematic, were never fatal.
- The ML reference specification was a workable idea, although discarded for later editions. In retrospect, it might have been better to start with a reference specification for ES3.

Douglas Crockford [2008c], in a blog post, attributed the failure of $ES4_2$ to excessive unproven innovation:

> It turns out that standard[s] bodies are not good places to innovate. That's what laboratories and startups are for. Standards must be drafted by consensus. Standards must be free of controversy. If a feature is too murky to produce a consensus, then it should not be a candidate for standardization. It is for a good reason that

[76]Lacking a written record, recollections are fuzzy regarding whether Adobe was represented by Smith, McAllister, or both.

> "design by committee" is a pejorative. Standards bodies should not be in the business of design. They should stick to careful specification, which is important and difficult work.

Allen Wirfs-Brock recalls feeling relief when Adobe announced its withdrawal from $ES4_2$. He was aware that the Microsoft executives responsible for Internet Explorer had come to understand that disinvestment in Internet Explorer had been a strategic mistake. IE was losing significant market share to Firefox and the executives were aware that Google was preparing to launch a new browser. Microsoft was actively confronting the perception among Web developers that it was opposed to technical advancement of the Web. Microsoft's opposition to $ES4_2$, particularly as exposed via the very public arguments involving Brendan Eich and Microsoft's Chris Wilson [Kanaracus 2007], was feeding that narrative. By June of 2008 Wirfs-Brock was worried that Microsoft might decide, for strictly business reasons, that it would be better to go along with $ES4_2$ rather than publicly oppose it.

The majority of the Oslo TC39 meeting [TC39 2008g] was spent explaining and socializing the concept of harmonization of TC39 around a common set of obtainable goals. The overall plan was to focus the entire committee on completing the ES3.1 release during 2009 while simultaneously collaborating in planning a more significant follow-on edition, code named "Harmony," that would not be constrained by the previous ten years of ES4 design decisions. There were discussions at the meeting about what features would or would not be "harmonious" but no serious objections to the basic plan were expressed either at the meeting or in post-meeting email discussions with TC39 members who were unable to attend the meeting. The steps of the plan were summarized in a white paper [Eich et al. 2008] prepared after the meeting:

1. Focus work on ES3.1 with full collaboration of all parties, and target two interoperable implementations by early next year.
2. Collaborate on the next step beyond ES3.1, which will include syntactic extensions more modest than current ES4 suggestions in both semantic and syntactic innovation.
3. Remove from consideration the ES4 concepts of "packages," "namespaces," and "early binding."
4. Rephrase other goals and ideas from ES4 to keep consensus in the committee; these include a notion of classes based on existing ES3 concepts combined with proposed ES3.1 extensions.

On August 13, Brendan Eich [2008b; Appendix M] emailed a slightly personalized version of the white paper to the es4-discuss mailing list. On August 19 Ecma International [2008] issued a short press release announcing that TC39 was going to focus its work on ES3.1. On August 15, Eich recorded a podcast [Openweb 2008] where he explained his view of the technical and pragmatic causes of $ES4_2$'s failure and his hopes for a harmonious future within TC39. Early in the podcast he said "the attempt to unify early binding and late binding through namespaces has failed." Later he elaborated:

> First packages were cut by us, ES4, we cut that. Second namespaces were cut by us, ES4, we cut that. We didn't do it to curry favor with 3.1. We did it because of problems with namespaces
> …
> this isn't a concession or a us versus them—this [$ES4_2$] is really a good attempt to try to unify things, going back to the Waldemar [Horwat] specs (or maybe even Common Lisp) to do with namespaces and packages and realizing they weren't right for the Web.

19 INTERLUDE: TAKING JAVASCRIPT SERIOUSLY

Starting in the late 1990s, TC39 members tried to redesign JavaScript as a language for serious professional programmers. By the late 2000s, the developers of browsers and other related platforms finally realized that JavaScript was a serious part of their platforms that needed serious engineering attention.

19.1 The JavaScript Performance Revolution

Performance was neither a concern nor a goal when Brendan Eich constructed Mocha in May 1995. There were not yet any JavaScript programs in existence and the anticipated programs were expected to be simple orchestrations of objects implemented with other, more efficient languages. JavaScript was not envisioned as a language for coding even moderately complex algorithms. Early JavaScript engines used either simple bytecode interpreters or parse tree evaluators to directly interpret JavaScript functions and utilized simple memory management schemes. They did not utilize any of the sophisticated high performance implementation techniques that had been developed in the 1980s and early 1990s for Lisp, Smalltalk, Self, and other dynamic languages. The basic architecture of Netscape/Mozilla's SpiderMonkey and Microsoft's JScript engines remained basically unchanged for ten years. New ES3-level language features were added and security issues were addressed but whatever performance gains were seen over that period can be attributable to hardware performance improvements driven by Moore's law [Moore 1975]. For most of that period maintenance of a browser's JavaScript engine was part-time work for a single software developer.

During the first half of the 2000s, the emergence of large AJAX-style JavaScript-based Web applications began to seriously push against the performance limitations of those first generation engines. By 2006–2007, Web developers were becoming more vocal about performance issues and browser vendors were starting to staff teams to address the performance limitations of their JavaScript engines. Being able to measure performance is an important starting point for improving performance and Apple's *WebKit*[g] team created the SunSpider JavaScript benchmark suite [Stachowiak 2007a] for that purpose. SunSpider was far from perfect and consisted of relatively small test cases, but it was derived from actual Web application code. Within a short period after its release the Web application developer community was routinely using SunSpider to compare browser JavaScript performance and talking about the results. Browser game theory generally prevents browser vendors from competing on the basis of JavaScript features, but they could and did begin to compete on JavaScript performance.

Different vendors took different paths to achieving high-performance JavaScript engines. In 2006, Google started developing what would ultimately become the *Chrome*[g] browser. Lars Bak led the development of Chrome's *V8*[g] JavaScript engine which built upon techniques he learned developing Smalltalk, Self, and Java virtual machines [Google 2008b]. When Chrome was released in September 2008 it became the new baseline for good JavaScript performance. One contemporaneous report [Hobbs 2008] found that V8 ran Google's benchmarks [Google 2008a] approximately 10 times faster[77] than SpiderMonkey in the then-current release version of Firefox. However, on the SunSpider benchmarks V8 was approximately only two times faster.

Mozilla's initial approach, called TraceMonkey [Gal et al. 2009], was based on the graduate work of Andreas Gal at University of California, Irvine. It used the existing SpiderMonkey interpreter augmented with a trace-driven code-specializing JIT compiler that generated optimized native code for dynamically identified execution hotspots. Apple's SquirrelFish Extreme [Stachowiak 2008a], also known as Nitro, used techniques inspired by Self and high-performance Lua implementations.

[77] In August 2018 one of the authors ran the same benchmarks in a browser using the then-current version of V8 on a 2011-vintage iMac. The reported result was approximately twenty times faster than the Hobbs' 2008 V8 results.

CommonJS Modules

```
// moda.js - source

var modp = require("modp");
exports.n = modp.p++;
exports.modName = "prefix"+exports.n;


// modb.js - source

var modx = require(require("moda").modName);
var propName = Object.keys(modx)[0];
exports[propName] = modx[propName];
```

Translates Into

```
// moda.js - CJS expansion
(function(exports,require,module){
  var modp = require("modp");
  exports.n = modp.p++;
  exports.modName = "prefix"+exports.n;
});

// modb.js - CJS expansion
(function(exports,require,module){
  var modx = require(require("moda").modName);
  var propName = Object.keys(modx)[0];
  exports[propName] = modx[propName];
});
```

Fig. 28. CommonJS modules are transformed by the module loader into functions implementing the module pattern. Sharing among modules is via the properties of dynamically constructed `exports` objects.

Microsoft initially tried to incrementally redesign their legacy JScript engine for use in IE 8, but for IE 9 they built Chakra, a completely new JIT-based JavaScript engine [Niyogi 2010].

All of these efforts were just the starting point for ongoing work to optimize JavaScript performance. Today, each major browser's development effort includes a substantial JavaScript team focusing on performance as well as security and new language features of the ECMAScript standard. Each of the engines developed by these teams is released under a compatible open-source license so the teams are able to build upon each other's work, sharing ideas and, sometimes, complete subsystems as they compete to produce the fastest JavaScript implementation.

19.2 CommonJS and Node.js

From its inception, JavaScript has also been hosted on server platforms to provide basic scripting capabilities. However, each platform was unique, providing its own distinct JavaScript APIs. For the first fifteen years of JavaScript's existence there was no common domain-independent, interoperable environment for non-browser JavaScript applications. In January 2009, Kevin Dangoor, a developer for Khan Academy, who had previously worked for Adobe and Mozilla, decided it was time to change that. He wrote a blog post [Dangoor 2009] describing the problems and invited the server-side JavaScript community to engage in solving the problems via an online discussion group and wiki. A year later in a follow-up blog post [Dangoor 2010] he summarized what he originally hoped to create as follows:

- A module system,
- A cross-interpreter standard library,
- A few standard interfaces,
- A package system, and
- A package repository

Within its first week, 224 members joined the discussion group [Kowal 2009a] and many of them expressed interest in contributing to the project. The initiative was initially called ServerJS but in August 2009 it was renamed to *CommonJS*[g] because its technologies would have applicability beyond servers. The initiative was focused on writing specifications rather than implementations.

By April 2009, the group had an initial module specification [CommonJS Project 2009]. The CommonJS Modules specification was based upon a design by Kris Kowal and Ihab Awad [2009a].

A CommonJS module is a JavaScript function body whose scope includes several bindings which enables the body code to interact with other modules. This is implemented by a synchronous module loader which fetches a module's source code, wraps the code with a skeleton function definition, and then parses and calls the synthesized function to initialize the module and its connections to other modules. As illustrated in Figure 28, module-scoped declarations become local variables of the synthesized function, and the control hooks of the system are exposed as function parameters whose values are provided by the loader. The `require` parameter is a function that synchronously performs a contextualized loading process for a requested module and returns its `exports` value.[78] By default the `exports` value is an object provided by the loader. Module code exports values by creating properties on the `exports` object. This is a dynamic run-time process. The module names, the actual `exports` value, and its property names and values may all be dynamically generated. This makes it difficult and sometimes impossible to predetermine an application's required modules and which entities are shared among them.

One of the early adopters of CommonJS Modules was *Node.js*[g] which was being developed by Ryan Dahl in early 2009. Node.js was conceived as an open-source JavaScript-based platform for building server applications which would be capable of handling a large number of simultaneous client connections. Node.js provided a JavaScript programming environment with libraries exposing a pervasive asynchronous I/O model. It combined common POSIX interfaces with JavaScript callbacks and a simplified version of the browser event loop. Its implementation consisted of Google's V8 JavaScript engine wrapped for standalone use, a CommonJS Module loader, and a set of C-implemented modules providing non-blocking versions of the POSIX APIs and other higher level file and network operations. The first public version was released in May 2009 [Node Project 2009], but it got significant attention only after Dahl [2009] gave a presentation at jsconf.eu in November 2009. Shortly after that Dahl was hired by Joyent which managed and supported further development of Node.js until responsibility was transferred to a foundation in 2015 [Node Foundation 2018].

Node.js was conceived as a technology for building server applications, but it became a platform that enabled JavaScript to be used as a general-purpose programming language on a wide variety of platforms including small embedded devices. The Node.js I/O modules combined with the high-performance V8 engine was comparable in capability and often superior in performance to other dynamic application languages such as Python and Ruby. It became the de facto standard way of writing command-line JavaScript applications. Node.js enabled Web programmers who had mastered JavaScript to transfer their skills to other kinds of applications and non-browser environments. Originally, developers of client Web applications programmed in JavaScript because there was no alternative. Many Node.js developers chose to use it because they preferred to program in JavaScript.

19.3 JavaScript: The Browser Universal Runtime

JavaScript is the programming language that is part of the suite of standards that defines the interoperable browser platform. It is the only language that developers of Web pages can expect to be available[79] in every browser. Other languages environments such as Java, Adobe Flash, and Microsoft *Silverlight*[g] are not part of this standard platform and must be integrated into a browser using a browser-specific add-on mechanism—if that browser is supported by that language.

[78] Only the first `require` for a specific module performs the full load process. The `exports` value is retained by the loader and immediately returned by subsequent `require` requests for the same module.

[79] Web page developers should still consider the possibility that the user of a browser has disabled JavaScript or that the page is being processed by a program that does not include JavaScript support.

Typically a language engine has to be separately installed by the browser user and may not be fully integrated into the browser's standard services such as the DOM-based graphics model.

Browser Game Theory predicts that the likelihood of success is extremely low for any attempt to extend the standard browser platform by adding another programming language. It takes a large investment for a browser vendor to design, implement, and promote a new language for the Web with no guarantee that it will find significant adoption by Web developers. Adoption requires that all major browsers agree to support a language that was designed by a competitor, has a small or non-existent user base, and which will become a perpetual maintenance burden. For example, in 2011 Google introduced the *Dart*[g] language and promoted it as a better programming language for the Web [Krill 2011]. Google distributed an experimental version of *Chromium*[g], [Google 2012a] the open-source foundation of their Chrome browser, which included a Dart virtual machine [Google 2012b] but it was never incorporated into a production version of Chrome nor any other browser.

With the emergence of AJAX/Web-2.0–style applications in 2005, Web developers started to write larger, more complex Web applications and some of them were looking for a programming language that seemed more suited to such applications than ES3-level JavaScript. What does a developer do if they need to write code that will run as part of a Web page in any browser and they need or want to write the code in a language other than JavaScript? The only alternative is to somehow use JavaScript to provide the runtime-support for the alternative language. This might be done by writing an interpreter in JavaScript for the alternative language. But in the mid-2000s, JavaScript engines were still implemented as relatively slow interpreters, and JavaScript was not a particularly good language for writing efficient interpreters. A doubly slow interpreted interpreter was not a very attractive solution. A more plausible way to host an alternative language was via a source-to-source translator—a compiler that translates the source code of the alternative language into JavaScript code that can run natively on a browser's JavaScript engine. Runtime performance of programs compiled in that manner could be relatively close to hand-written JavaScript if there is a reasonably close match between the semantics of the alternative language and JavaScript's semantics.

Google Web Toolkit (GWT) [Google 2006], publicly released in May 2006, was the first widely used AJAX toolkit using source-to-source translation; GWT incorporated a Java to JavaScript compiler. It was successfully used for a number of significant Google public-facing Web applications and also found significant use outside of Google. The success of GWT proved the feasibility of targeting JavaScript for source-to-source translation and translators for many other languages followed. A January 2011 list of languages that compile to JS [Ashkenas et al. 2011] has nineteen entries. A 2018 version of the same list [Ashkenas et al. 2018] includes more than 270 languages which are translated to or otherwise hosted by JavaScript. Some of these are toys or incomplete implementations. However, many are serious compilers with significant numbers of users. There is even a Dart compiler that targets JavaScript.

Source-to-source translation was used not only for supporting legacy languages on Web pages. It also provided a means for experimenting with new languages and for extending JavaScript. One of the most successful source-to-source translators was *CoffeeScript*[g] [Ashkenas 2010], developed by Jeremy Ashkenas in 2009 and 2010. Before becoming a Web developer, Ashkenas had programmed in the Ruby language and preferred Ruby's relatively punctuation-free syntax and Python-style significant whitespace to the C-style syntax used by JavaScript. He created CoffeeScript as new syntactic skin for JavaScript while keeping the underlying JavaScript runtime semantics. Ashkenas [2009] announced his work on CoffeeScript with this description:

> JavaScript has always had a gorgeous object model hidden within Java-esque syntax. CoffeeScript is an attempt to expose the good parts of JavaScript through syntax that

favors expressions over statements, cuts down on punctuation noise, and provides pretty function literals. This CoffeeScript:

```
square: x => x * x.
```

Compiles into this JavaScript:

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

In addition to "pretty functions," CoffeeScript introduced a number of syntactic programming conveniences including class declarations and destructuring operations which easily translated into JavaScript code. A number of the CoffeeScript features were similar to features that were being contemplated for ECMAScript Harmony. CoffeeScript validated that JavaScript programmers were interested in such features. CoffeeScript quickly became quite popular and was adopted by a number of major website developers. Its usage waned after ES2015 became widely available.

At the May 2011 JSConf, Brendan Eich shared the stage with Jeremy Ashkenas and spoke about CoffeeScript and its role in the Harmony evolution of JavaScript. In his presentation, Eich [2011c] introduced a Word of the Day, "transpiler[g]," to describe source-to-source compilers like CoffeeScript. This was not the first time that the term "transpiler" was coined with that usage, but it was not widely known or used before Eich's talk. Subsequently it came into common usage within the JavaScript developer community and beyond.

Alon Zakai's [2011] Emscripten is a transpiler that translates C/C++ into efficient JavaScript code. It is based upon the observation that JavaScript's 32-bit arithmetic coding patterns (§3.7.3) and binary TypedArray data structures could be used to define a C execution environment that was easily optimized by JIT-based JavaScript engines. Emscripten inspired asm.js [Herman et al. 2014], which is a specification defining a set of JavaScript code-patterns that such compilers should generate and that should be recognized and optimized by engines. The success of asm.js led to development of WebAssembly [Haas et al. 2017], which extends JS engines with a bytecode-level interface that can be used as a compilation target for C/C++ and similar low-level languages.

Part 4: Modernizing JavaScript

20 DEVELOPING ES3.1/ES5

Throughout most of 2007, the $ES4_2$ working group believed that the ES3.1 effort was simply a competitive attempt to derail $ES4_2$ and that it lacked any technical substance. However, Douglas Crockford, Pratap Lakshman, and Allen Wirfs-Brock were committed to developing an incremental improvement to the ES3 specification that brought it up to date and corrected sources of interoperability issues. The first step after posting the initial goals, design principles, and proposed feature-level changes [Lakshman et al. 2007] for ES3.1 was to develop a fuller understanding of the then-current state of JavaScript in Web browsers and how Web Reality differed from the ES3 specification.

An immediate concern for the ES3.1 working group was that Microsoft's JScript implementation for Internet Explorer had a reputation of not being in compliance with Web standards. In order to understand the validity and scope of those concerns regarding ECMAScript, Allen Wirfs-Brock asked Pratap Lakshman to do an analysis to determine all of the ways that IE JScript deviated from ES3. This resulted in an 87-page "JScript Deviations from ES3" report [Lakshman 2007c] which was completed in September 2007. The report had three major sections. The first major section identified each place the then-current JScript implementation deviated from a clear requirement of the ES3 specification. For each deviation the report provided the ES3 specification language that

2.15 String.prototype.split: §15.4.4.14
ES3 states that "If separator is a regular expression that contains capturing parentheses, then each time separator is matched the results (including any undefined results) of the capturing parentheses are spliced into the output array."

JScript ignores the capturing parentheses. FF outputs empty strings instead of undefined.
Example:

```
<script>
alert("A<B>bold</B>and<CODE>coded</CODE>".split(/<(\/)?([^<>]+)>/));
</script>
```

Output:
IE: `A,bold,and,coded`
FF: `A,,B,bold,/,B,and,,CODE,coded,/,CODE,`
Opera: same as FF
Safari: same as IE

Fig. 29. An ES3 deviation as documented in JScript Deviations Report [Lakshman 2007c]

was violated, a test case that could be used to observe the deviations, and the results of running the test on the then-current releases of Internet Explorer, Mozilla Firefox, Opera, and Apple Safari. Those browsers were considered the "top four" browsers as of that time. Figure 29 provides an example of the sort of deviations that were identified. Some deviations were unique to Internet Explorer, some occurred identically in all tested browsers, and some occurred identically in Internet Explorer and in one or two of the other browsers.

The second major section of the deviations report identified all places in the ES3 specification where behavior was explicitly defined as being implementation dependent or was inadequately defined. This section also provided test cases and the results of running the test on the four major browsers. The final major section described features implemented by Internet Explorer that were extensions to the ES3 specifications. Wirfs-Brock [2007b] also prepared lists of documented Firefox extensions to ES3. Douglas Crockford and Allen Wirfs-Brock met August 16, 2007, to review drafts of these documents. The result of the meeting was a set of tentative changes [Wirfs-Brock and Crockford 2007] to be made in the ES3.1 specification.

The development of ES3.1 got seriously underway at the January 2008 TC39 meeting, where the goals were reviewed and several additional TC39 participants expressed an interest in working on it. On February 11, Lakshman sent an ES3.1 call-to-action message to the TC39 private email list. The email called attention to the deviation and interoperability documents that had been prepared the previous summer and requested more feedback on them. On February 21, a conference call was held at which a work schedule of twice-weekly conference calls was established. The participation in those calls was considerably greater than in previous ES3.1 discussions. Figure 30 lists regular participants. Initially, direct emails were used to exchange and discuss proposals. Some additional ES3.1 discussions occured on the es4-discuss email forum. However, the volume of traffic relating to $ES4_2$ topics made it hard to pick out the ES3.1-specific topics, so in April a separate es3.1-discuss[80] email forum [TC39 et al. 2008] was created and most of the ES3.1 design discussions between meetings moved to it.

One of the first topics of discussion [TC39 2008d] was a review of the overall goals for ES3.1 and the design rules that would be followed in resolving issues and adding new features. An early position that had been advocated by developers from the Microsoft Live team and several

[80]In March 2009, this email forum was renamed to es5-discuss.

Douglas Crockford	Yahoo!
Pratap Lakshman	Microsoft
Mark S. Miller	Google
Adam Peller	IBM
Sam Ruby	IBM
Allen Wirfs-Brock	Microsoft
Kris Zyp	The Dojo Foundation

Fig. 30. 2008 ES3.1 WG Meetings Regular Participants

other Web framework developers was to avoid any new syntactic extensions that would cause scripts to fail to parse on existing or older versions of browsers. But a "no new syntax" rule was overly constraining and ignored the reality that various browsers already had some syntactic extensions. That discussion led to the "3 out of 4" rule based upon the four most-widely known browsers (Internet Explorer, Firefox, Opera, and Safari) that had been analyzed in Microsoft's JScript Deviations document. When 3 out of 4 of those browsers were in agreement on a feature or had a common behavior, that agreement should be adopted by the ES3.1 specification. This rule led to a broader discussion of how ES3.1 should approach browser interoperability issues.

There was agreement that an overriding ES3.1 principle should be "don't break the Web" by specifying language changes which would alter the behavior of existing Web pages that already interoperated across major browsers. But there were hundreds of millions of existing Web pages. Which aspects of the ECMAScript specification did they actually depend upon? Which changes would be Web breaking? Anecdotal reports from browser implementors suggested that, because of the massive base of existing Web pages, every interoperable browser feature (no matter how obscure or implausible the usage) was likely used by some existing pages. Based upon that view, features which were common to all four major browsers could not be changed, and features common to 3 out of 4 browsers were strong candidates for standardization. But what about features and behaviors that were common to only 2 out of 4 browsers or that differed among all browsers? Apparently such features and behaviors were not essential to the existing interoperable Web and could potentially be modified in the course of standardization.

The working group also observed that generally all allowances for implementation variability within the ECMAScript specification were hostile to the creation of interoperable Web pages. Situations where traditional language specifications might permit implementation-specific variation in order to provide flexibility for language implementors or to accommodate known variations among implementations were fundamentally incompatible with the idea of a worldwide interoperable Web accessible through multiple independently created Web browsers. The ECMAScript specification needed to be more prescriptive and detailed than traditional language specifications and, wherever possible, existing allowances for implementation variation needed to be eliminated. Following these initial discussions in February, Douglas Crockford [2008a] posted revised ES3.1 goals to the TC39 Wiki (Figure 31).

At the March 2008 face-to-face meeting, the working group agreed that it was important to immediately start writing the actual ES3.1 specification document. Patrap Lakshman had arrived at the meeting with a version of the ES3 specification to which he had made the corrections from the Mozilla-maintained ES3 errata [Horwat 2003b]. The working group agreed to use that as the ES3.1 base document and asked Lakshman to serve as the editor. Like the previous editions, the specification document would be composed using Microsoft Word. Change tracking relative to the 3rd Edition would be used to track the evolution of the specification for review and to ensure

1. Browser implementation unification: Consider adopting features that are already implemented in 3 of the 4 browser brands, or that are deployed in 3 out [of] 4 user computers and reduce cross browser incompatibilities.
2. ES3.1 shall improve the language to benefit casual developers by reducing confusing or troublesome constructs.
3. ES3.1 shall improve the language to benefit major websites by reducing confusing or troublesome constructs.
4. ES4 shall become a superset of ES3.1.
5. ES3.1 shall be a friendly base for a secure subset.
6. ES3.1 shall attempt to correct errors in ES3.
7. ES3.1 new features shall require concrete demonstrations.
8. ES3.1 may deprecate (or eliminate with opt-in) features that are problematic for performance, security, or reliability.
9. ES3.1 shall provide virtualizability, allowing for host object emulation.

Fig. 31. February 2008 ES3.1 Revised Goals [Crockford 2008a]

Lakshman	New Array methods based upon Mozilla "Array Extras" plus `reduce` and `reduceRight`
Lakshman	Add support for array-like string indexing
Lakshman	`Date` improvements
Lakshman	Strict mode property access semantics
Crockford	JSON support
Crockford	Unicode update
Peller	Recommend changes based on Microsoft's deviations document
Ruby	Decimal arithmetic
Zyp	Syntactic getters/setters in object literals
Wirfs-Brock	Static methods for property creation and inspection
Wirfs-Brock	Update pseudo code notation and conventions
Miller	Object freeze/seal and review everything from a security perspective

Fig. 32. ES3.1 Working Group Task Assignments as of March 28, 2008. [TC39 2008c]

that changes could be reintegrated with the $ES4_2$ effort. Members of the working group were assigned (Figure 32) to develop specification text for specific new features. As they were completed, Lakshman would integrate their work into the master draft.

On May 29, 2008, Pratap Lakshman posted to the TC39 wiki the initial draft of the ES3.1 specification. Updated drafts were typically posted weekly with a "review draft" posted two to three weeks before each scheduled TC39 meeting. A total of 26 intermediate drafts were posted between May 29, 2008, and March 2, 2009.

IBM had long advocated that JavaScript needed to support decimal arithmetic. Starting at the November 19, 1998, TC39 working group meeting, Mike Cowlishaw had argued for its inclusion in ES3 and in $ES4_1$. When IBM reëngaged with TC39 to contribute to $ES4_2$ and ES3.1 they again strongly advocated for the inclusion of decimal support. The IBM participants made sure that TC39 was aware that it was IBM's policy to oppose all new language standards which did not include support of decimal arithmetic. Many in TC39 were skeptical of the feasibility of accomplishing that, but Brendan Eich was supportive of IBM and pointed out that the most common bugs reported against Firefox were from JavaScript developers who did not understand the semantics of binary floating point arithmetic. Eich helped Sam Ruby get started developing a prototype, using Mozilla's SpiderMonkey engine, that implemented IEEE 754-2008 decimal floating point as a new primitive

data type which could be used combination with the Number type in mixed-mode expressions. A fairly complete specification of this decimal feature was incorporated into the September 2008 and November 2008 ES3.1 drafts. The intent of the November 19–20, 2008, TC39 meeting was to make the final decisions regarding which new features to retain or remove from the ES3.1 draft. The first item considered was decimal arithmetic support. The committee's conclusion was that the decimal design was still too immature and had remaining design issues which were unlikely to be resolved without delaying ES3.1. The concerns were documented in the meeting minutes [TC39 2008a] and concluded:

> Because of these concerns the decision was made to defer inclusion of decimal support until the Harmony revision of ECMAScript. The attendees acknowledged that very significant progress has been made in the development of the ECMA-Script decimal proposal and want to thank Sam Ruby of IBM for the effort he has put into its development. The attendees encourage the continued development of the proposal by Sam and other TC39 members and are optimistic that a fully integrated and generic version of decimal arithmetic can become an integral part of the Harmony revision.

The decimal materials were absent from the next review draft released in January 2009.

At the March 25–26, 2009, meeting [TC39 2009d] Pratap Lakshman announced that he was resigning as ECMA-262 editor because Microsoft was transferring responsibility for JavaScript development to a new Redmond-based group, and he had declined the opportunity to relocate with it. The committee appointed Allen Wirfs-Brock to succeed him as editor.

Wirfs-Brock recalls that at a break during that TC39 meeting he approached Brendan Eich and suggested that ES3.1 should be rechristened with a whole-number edition designation. The argument for the new designation was that E3.1 had grown to be a full-fledged revision of ECMA-262, as significant as the three previous editions. Because of the amount of publicity that the discontinued $ES4_2$ work had received, designating ES3.1 as the 4th edition would cause confusion for both the JavaScript developer community and Web search engines. Instead, Wirfs-Brock suggested that Ecma should permanently retire the ECMA-262 4th Edition designation and release the ES3.1 work as the 5th Edition. Eich agreed, so when the meeting resumed they presented the idea to the entire committee which accepted it. The committee also agreed to accept the then-current draft, after updates agreed to at the meeting, as the final draft. On April 7, 2009, the "final draft" was released with the 5th Edition designation [Lakshman et al. 2009]. That final draft was followed by five release-candidate drafts containing minor technical and editorial corrections. In August 2009, Apple discovered [Hunt 2009] that the decision to make `arguments` objects inherit from `Array.prototype` had an unanticipated interaction with the Prototype framework that broke several Apple websites and the NASA website. That change was removed from the final specification.

On September 23, 2009, TC39 [2009b] voted to accept completion of ES5 and to forward it to the Ecma General Assembly for its approval. The final draft for Ecma GA review and approval was posted October 28, 2009. *ECMA-262 5th Edition* was approved by the General Assembly [Ecma International 2009a] December 3, 2009, ten years after the approval of the 3rd Edition. The GA vote was 19 for and 2 opposed. IBM voted no because the standard did not include support for decimal arithmetic. Intel stated their no vote simply reflected that they had not had sufficient time to do a complete intellectual property review of the specification.

ECMA-262 5th Edition was submitted as a fast-track revision of the ISO/IEC ECMAScript standard. It went through the ISO national bodies review process and based upon that feedback, Allen Wirfs-Brock incorporated a number of editorial corrections and clarifications into the specification. That revision was published in June 2011 as *ECMA-262 Edition 5.1* and as *ISO/IEC 16262 Edition 3*.

20.1 ES5 Technical Design

Even though the original ES3.1 goals were very modest ES5 includes several technical innovations.

20.1.1 Strict Mode. ES5 strict mode is the direct end product of Douglas Crockford's goal of "correcting mistakes and inconveniences" in JavaScript's design. A few of the inconveniences, such as the inability to use reserved words as property keys in object literals and after a dot were then-currently syntax errors and could be corrected in ES5 without affecting existing code. However, many of JavaScript's misfeatures could not be unconditionally fixed because they would be changes that could change the runtime behavior of existing code and hence "break the Web." The idea for strict mode was to give JavaScript developers an opportunity, in new or updated code, to explicitly opt-in to a dialect of the language that incorporated such fixes. Browsers, would have to support both strict mode and the legacy non-strict code and ideally strict mode should be selectable at the individual function level so that existing scripts could be incrementally converted to using strict mode. The hope was that over time strict mode would become the dominant dialect for writing new code. However, initial adoption was a concern. It was assumed that there might be a considerable delay before ES5 strict mode would be implemented by all major browsers. Browser game theory predicted that if opting into strict mode made scripts unusable on some popular browsers, then developers would not use it. This problem was avoided by making strict mode subtractive. It does not add new features to ECMAScript; instead it removes problematic features. Bug-free strict mode code when run on a browser that did not support strict mode should continue to work as its developer expected.

An early issue was how the opt-in to strict mode would work. Fine-grained selection of strict mode required that the opt-in be via a mechanism that could be easily embedded within a script. It could not be external, such as a `<script>` element attribute. The ES4 efforts had contemplated a `use` directive that could be placed within ECMAScript code to select various modes. But such a directive would violate the ES3.1 "no new syntax" design rule. One possibility was to use a special form of comment as a directive. However, the ES3.1 working group was reluctant to make comments, of any form, semantically significant because JavaScript minimizers remove comments. Allen Wirfs-Brock observed that the syntax of an ECMAScript *ExpressionStatement* makes any expression, including those that consist of only a literal string constant, into a valid statement as long as it is explicitly or implicitly (via ASI) followed by a semicolon. That means that a statement such as `"use strict";` is syntactically valid ES3 code. Because it is simply a constant value, evaluating it has no side effects in ES3. It is a *no-op*[g]. It appeared quite safe to use such a statement as the opt-in for strict mode as it seemed highly unlikely that any existing JavaScript code would already have used that exact statement form and an ES3 implementation would ignore its presence in any ES5 code that was loaded. The working group adopted that idea. A statement of the form `"use strict";` occurring as the first statement of a script or a function body indicated that the entire script or function should be processed using strict mode semantics.

One of the main goals of strict mode was to explicitly catch coding errors that were easy to make but not obvious at runtime. Strict mode adds the following new runtime errors:

- Assignment to an undeclared identifier. In legacy JavaScript an assignment to a mistyped variable name results in creation of a property of on the global object.
- Assignment to a read-only own or inherited property. In legacy JavaScript this silently does nothing.
- Attempting to create a property on a non-extensible object. Such objects did not exist prior to ES5, but for legacy consistency doing this outside of strict mode in ES5 silently does nothing.
- Applying the `delete` operator to a non-deletable property. In legacy JavaScript `delete` would return false.

- Applying the delete operator to a variable reference produces a syntax error. In legacy JavaScript `delete` returns false for explicitly declared variables. If the variable reference is backed by an object via a `with` statement or is a property of the global object it is deleted in legacy JavaScript.

Strict mode also removes or modifies features that may make programs more confusing, harder to optimize, or less secure:

- The `with` statement is disallowed. A `with` statement provides a form of dynamic scoping of variable references which can be confusing and is difficult for implementations to optimize.
- The `eval` function cannot be used to dynamically add new bindings to the current scope.
- The names `eval` and `arguments` cannot be used as variable or parameter names.
- A function's `arguments` object is not joined (§3.7.5) to its formal parameters. Instead, a strict mode `arguments` object is an array-like object whose elements are a snapshot of the argument values passed to the function. Modifying an element does not modify the value of the corresponding formal parameter and vice versa.
- The `arguments` object of a strict mode function does not have a `callee` property (§5). Passing such an object to other code no longer implicitly transfers the ability to call its function.[81]
- An implementation is forbidden from providing a `caller` property (§3.7.5) on the `arguments` object of a strict mode function. A `caller` property was a non-standard but widely implemented extension to ES3 which enabled walking a function's call stack and retrieving the calling functions.
- Calling a strict mode function without providing a `this` value does not make the global object available to the function (§3.7.4).

Other features on Douglas Crockford's [2007d] list of mistakes and inconveniences were considered for strict mode but not included. For each feature, either TC39 could not reach consensus that the feature was undesirable or it was discovered that the change would not be subtractive. For example, while Crockford and many others disliked JavaScript's Automatic Semicolon Insertions, many developers preferred to code without explicit semicolons. Also, changing the meaning of `typeof null` to return something other than `"object"` would not be subtractive.

20.1.2 Getters, Setters, Object Meta Operations. Beginning with the first implementations of JavaScript, some properties of certain built-in and host-provided objects had special characteristics that were not available for objects created using JavaScript code. For example, some properties had read-only values or could not be deleted using the `delete` operator, and method properties of built-in and host objects are skipped when enumerating properties using the `for-in` statement. In ES1, these special semantics were specified by associating ReadOnly, DontDelete, and DontEnum attributes with the specification's model of object properties. These attributes are tested by the pseudocode which defines the semantics of language features which are sensitive to them. These attributes are not reified—there were no language features that enabled JavaScript code to set these attributes for either newly created or preëxisting properties. ES3 added an `Object.prototype.propertyIsEnumerable` method for testing for the presence of the DontEnum attribute, but there were no corresponding methods for non-destructively testing for the ReadOnly or DontDelete attributes. Similarly, many of the host objects provided by the browser DOM expose properties, typically called "getter/setter properties" but christened "accessor properties" by ES5, which perform computations when the value of the property is set or retrieved. The lack of standardized support for these features made it impossible for JavaScript programmers to define

[81] That other code may come from an unknown source and may not be trustworthy.

libraries that follow the same conventions as the built-in or host objects or to implement polyfills which faithfully emulate such objects.

The unified solution to these problems is the largest collections of new ES5 functionality. The feature set does not have an official name but is informally called the "Static Object Functions"[82]" or "Object Reflection Functions." Allen Wirfs-Brock [2008] wrote a design rationale document for this feature set. It presents use cases and also includes these design guidelines that were followed:

- Cleanly separate the meta and application layers.
- Try to minimize the API surface area such as the number of methods and the complexity of their arguments.
- Focus on usability in naming and parameter design.
- Try to repeatedly apply basic elements of a design.[83]
- If possible, enable programmers or implementations to statically optimize uses of the API.

The first guideline discouraged adding additional methods such as `propertyIsEnumerable` to `Object.prototype` which would further blur the separation of the meta and application layers. Instead, the ES5 working group decided that such functions should be segregated from application objects by making them properties of a namespace object. The working group considered adding a new built-in global object called `Reflect` to serve as the namespace object but they were concerned about possible name conflicts with existing code. Ultimately, they decided to expose the new functions as properties of the `Object` constructor rather than as properties of `Object.prototype`. The `Object` constructor was a good candidate to use as a namespace because it was a preëxisting global on which implementations and previous editions of the standard had not specified any properties; also, its name aligned with the idea of reflecting upon the definition of objects.

The next issue was determining the form of the API. Following the second guideline, the ES5 designers wanted to avoid separate query and assignment functions for each property attribute or for setting and retrieving the functions performed by accessor properties. The designers considered various ways to combine this functionality into a small number of functions. Some possibilities included a single function with bit encodings of Boolean attributes, such as "read-only" or a single function with a large number of positional parameters. However, both of those approaches had poor usability. Using optional keyword arguments might have solved those usability issues but ES5 lacked keyword arguments.

Allen Wirfs-Brock suggested using a descriptor object whose properties would correspond to the various property attributes. Such descriptors could be used for both defining and inspecting properties. Wirfs-Brock's first draft proposal[84] showed an example of a possible API for adding a property to an object called `obj`:

```
Object.addProperty(obj, {name:"pi", value:3.14159, writable:false});
```

In this example, the descriptor is coded as an object literal, and default values are used for any properties missing from the descriptor corresponding to other property attributes. A hypothetical `defineProperty` function, taking a similar descriptor, could be used to change the attribute values of a preëxisting property. For `defineProperty`, attributes corresponding to absent descriptor properties would be left unmodified. Finally, a `getProperty` call could be used to obtain a complete descriptor for any preëxisting property of an object.

[82] Alternatively, "Methods." The distinction between an object used as a namespace and an object used as a behavioral abstraction is conceptual and not reflected in the actual semantics of the language. Some JavaScript programmers use the term "method" to make that distinction, others do not.

[83] Meaning, individual features should be designed using a set of common concepts and syntactic elements.

[84] This proposal is not directly available, but Miller's [2008a] response embeds most of the text of the proposal.

Function Name	Behavior
`Object.create`	Create a new object using a provided object as its prototype. Optionally, add the properties defined by a property map.
`Object.defineProperty`	Based on a property descriptor, create a new property or update the definition of an existing property
`Object.defineProperties`	Create or update the definition of the properties as specified in a property map.
`Object.getOwnPropertyDescriptor`	Return the descriptor object of a named property if it exists or `undefined` if it does not exist.
`Object.getOwnPropertyNames`	Return an `Array` containing the string names of all the own properties of an object.
`Object.getPrototypeOf`	Return the prototype object of the argument object.
`Object.keys`	Return an `Array` containing the string names of an object's own properties which are visible using `for-in`.
`Object.preventExtensions`	Prevent any more properties from being added to an object.
`Object.seal`	Prevent adding any more properties or changing the definition of an object's own properties.
`Object.freeze`	Seal an object and freeze the values of all its own data properties.
`Object.isExtensible`	Test if additional own properties may be added.
`Object.isSealed`	Test if an object is sealed.
`Object.isFrozen`	Test if an object is frozen.

Fig. 33. ES5 Object Reflection Functions

Mark Miller improved on the proposal by suggesting that `defineProperty` could be defined to support both the "add new property" and "modify existing property" use cases. Miller also suggested removing the name property from property descriptors and instead wrapping descriptors in an enclosing object whose property names were the names of the affected properties in the target object. Such a "property map" would allow multiple properties to be defined using a single call. For example, the following operation defines properties named `x` and `y`:

```
Object.defineProperties(obj, {
  x: {value: 0, writable: true},
  y: {value: 0, writable: true}
});
```

ES5

Miller proposed eliminating `defineProperty` and having only the `defineProperties` form as it could easily be used for a single property. However, that formulation made it difficult to define a property that had a computed name. ES3.1 did not have a syntactic way to place a computed value in a property name position of an object literal. Ultimately, ES3.1 provided both `defineProperty` for defining a single property with the name passed as a separate argument and `defineProperties` which can define multiple properties using a property map. The complete set of Object Reflection Functions defined by ES5 is shown in Figure 33.

Accessor properties are supported via an alternative formulation of a property descriptor. An accessor property is defined using a descriptor that has either or both of the `get` and `set` attributes instead of a `value` attribute. For example, an accessor property that mediates access to a data property can be defined as:

```
Object.defineProperties(obj, {
  x: {set: function(value) {this.privateX = value},// public accessor property
      get: function() {return this.privateX}
  },
  privateX: {value: 0, writable: true}  // "private" data property
});
```

ES5

In addition to this reflection-based interface, ES3.1 also adds syntactic support for defining accessor properties using object literals. This feature was already present in three out of four different browsers, so it met the criteria for adding new syntax. An accessor property is defined by including within an object literal a function definition where the keyword `function` is replaced with either `get` or `set`, for example:

```
var obj = {
  privateX: 0,                          //a normal data property
  set x(value){this.privateX = value}, //accessor property x setter
  get x(){return this.privateX},        //accessor property x getter
  get negX(){return -this.privateX}     //a get-only accessor
  };
```

ES5

Supporting these new capabilities required extending the internal object model, first defined in the ES1 specification, and exposing parts of it via the object reflection API. This also provided an opportunity to reconsider the terminology of the object model. ES1 had described properties as having a value and a set of attributes. The ES1 attributes were ReadOnly, DontEnum, and DontDelete. The ES1 attributes were not stateful. They were markers that were attached to properties with meaning assigned to their presence or absence. The ES3.1 designers wanted to reify the attributes as properties of property descriptor objects. They accomplished this by changing the internal model such that the ES1 attributes were modeled as Boolean-valued state variables associated with each object property, and by reconceptualizing the property value as another state variable. The internal naming convention for attributes was changed to follow the double bracket pattern used for internal methods. The model was extended to include accessor properties by adding the attributes [[Get]] and [[Set]] whose values were respectively the getter and setter functions (or `undefined`, indicating the default function) which were invoked by value references and assignments to the property. A property could then be distinguished as either a data property or an accessor property by whether or not it had the [[Value]] attribute and neither of the [[Get]] nor [[Set]] attributes.

Support for accessor properties required updating the specification of the [[Get]], [[Put]], and [[CanPut]] internal methods originally defined by ES1. Support for the property descriptors used by the Object reflection API required the addition of [[DefineOwnProperty]], [[GetOwnProperty]], and [[GetProperty]] internal methods. But that reflection API was still insufficient. In ES3.1 the `for-in` statement's enumeration of property keys and the `Object.getOwnPropertyNames` and `Object.keys` functions still used informal prose to specify their semantics.

The final step in the design of the Object reflection API was deciding upon consistent and usable naming conventions for the vocabulary used to expose property attributes as property names within property descriptor objects. In particular, names like DontEnum and ReadOnly lacked internal consistency and raised usability concerns. This was particularly true when treated as Boolean-valued flags. For example, to make a property enumerable would be expressed as a double negative, setting DontEnum to false. Early in 2008, on an $ES4_2$-related thread Neil Mix [2008b] suggested that "enumerable," "writable," and "removable" (for DontDelete) were better names for

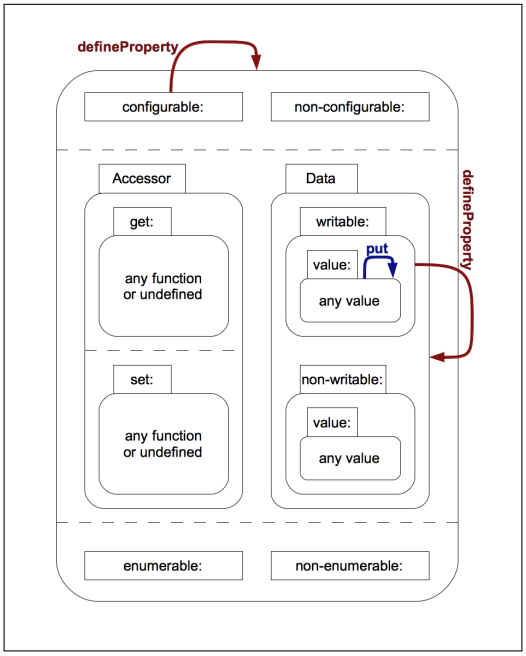


Fig. 34. ES5 Property Attribute Statechart [Miller 2010b]

the attributes. Mark Miller [2008b] responded with appreciation for those names and proposed a design guideline: attribute names should say what is allowed rather than what is denied. He further proposed following the security best practice of deny by default. When defining a property it would be necessary to explicitly enable any desired attributes.

The Object reflection API provided a new capability not available in previous versions of ECMAScript. It allowed a program to change the attributes of an existing property including changing a property between being a data property and an accessor property. Consideration was given to whether an additional attribute would be needed to disable the ability to make such changes. Possible attribute names considered included "dynamic," "flexible," and "fixed." There was concern about the possible impact upon existing implementations of adding an additional Boolean property attribute. What if an implementation did not have an extra bit available that could be used to represent that attribute? Eventually, the ES3.1 working group realized that changing the attribute of a property was equivalent to atomically querying the current attributes of the property, deleting the property, and then recreating a property with the same name but with modified attribute values. Given that equivalence, a single attribute could be used to enable both deletes and modifications. The DontDelete/removable attribute was renamed "configurable"[85] and assigned that meaning. A statechart [Harel 2007] diagram for the ES5 property attributes was drafted by Mark Miller [2010b; Figure 34] and posted on the ECMAScript wiki. Notice that when the configurable attribute is false it is still possible to change the writable attribute of a property from true to false. This anomaly was created so that a security *sandbox*[g] could change certain built-in properties from non-configurable but writable into non-configurable, non-writable.

Douglas Crockford was a proponent of using of the prototypal style of object-oriented programming for JavaScript applications. He had promoted the use of a function named `beget` for creating

[85]The term "configurable" was suggested by Neil Mix [2008a].

objects with an explicitly provided prototype. The ES5 function `Object.create` was essentially the `beget` function with a property map added as an optional second parameter, for example:

```
var point1 = beget(protoPoint); //Create a point , using Crockford style
point1.x = 0;
point1.y = 0;

var point2 = Object.create(protoPoint, { //using ES5 declarative style
  x: {value: 0},
  y: {value: 0}
});
```

ES3 with Crockford's `beget` function compared to ES5

Allen Wirfs-Brock had hoped that JavaScript programmers would adopt the declarative style and that implementations would recognize that pattern to optimize object creation. In practice, a usability issue prevented wide adoption of that ES5-enabled pattern. The problem was with the selection of default property attributes. Going back to JavaScript 1.0, a property created by implicit assignment had the equivalent of property attributes `{writable: true, enumerable: true, configurable: true}`. But the deny-by-default policy used in designing ES5 attribute descriptors meant that all of those attributes would have the default value `false` for the `Object.create` declarative-style example as shown in the following:

```
//Using Object.create in a Crockford style
var point1 = Object.create(protoPoint);
point1.x = 0;
point1.y = 0;
// point1.x attributes: writable:true, enumerable:true, configurable:true
// point1.x attributes: writable:true, enumerable:true, configurable:true

//using Object.create in a declarative style
var point2 = Object.create(protoPoint, {
  x: {value: 0},
  y: {value: 0}
});
// point2.x attributes: writable:false, enumerable:false, configurable:false
// point2.x attributes: writable:false, enumerable:false, configurable:false
```

ES5

To exactly match the effect of the `beget` example an ES5 JavaScript programmer would have to write:

```
//Create a point instance, with ES5 support and traditional attribute values
var point2 = Object.create(protoPoint, {
  x: {value: 0, writable:true, enumerable:true, configurable:true},
  y: {value: 0, writable:true, enumerable:true, configurable:true}
});
```

ES5

That formulation was too verbose for most programmers, who wanted to continue to use the more permissive defaults traditionally used by JavaScript. In practice, the single argument form of `Object.create` is routinely used to create new objects and `Object.defineProperties` is used to define and manipulate properties of previously created objects, but the two argument form of `Object.create` is seldom used to define the properties of new objects.

20.1.3 Object Integrity and Security Features. The HTML `<script>` element `src` attribute, introducted by Netscape 3, enabled a Web page to load JavaScript code from multiple Web servers. In its most common formulation, the scripts were loaded into a single JavaScript execution environment where they shared a global namespace. Cross-site scripts could directly interact—this enabled the creation of mashup applications. The ability to load cross-site scripts was widely used and was a key enabler of advertising-based Web business models. However, cross-site scripts could also tamper and interfere with each other and with scripts from the page's home site. Web developers ultimately realized that there was a risk that third party scripts could steal confidential user data such as passwords or modifying the behavior of a page to trick users. By 2007, malicious ads were starting to be observed, unknowingly distributed by Web advertising brokers. The browser vendors developed various HTML and HTTP level features such as Content Security Policy (CSP) to address the problem, but features at that level do not directly address the many low-level JavaScript vulnerabilities [Barth et al. 2009].

While they were participating in the ES3.1 working group, both Douglas Crockford [ADsafe 2007] and Mark Miller [Caja Project 2012; Miller et al. 2008] were actively developing technologies for providing JavaScript execution sandboxes that could be used to securely host the execution of untrusted third-party JavaScript code. While ES3.1's strong backward compatibility requirements meant that it was not going to be possible to eliminate many of the known third-party script vulnerabilities, both Crockford and Miller pushed for eliminating vulnerabilities that could be compatibly removed and for adding new features that would facilitate the creation of secure sandboxes. Mark Miller, in particular, was interested in features needed to build sandboxes based on object-capabilities [Miller 2006].

The biggest issue was the mutability of JavaScript objects. By default, every JavaScript object, including standard library objects, are completely mutable by any code that obtains a reference to the object. Properties, including methods, can be added, have their values changed, or be deleted. This applies to a directly referenced object and to all objects indirectly referenceable starting from a root object. Even though ES3 did not provide a way to directly modify an object's reference to its prototype object, all major browsers, except for Internet Explorer had implemented the non-standard property `__proto__` which could be used to modify an object's prototype inheritance chain. The only exceptions to this pervasive mutability were a small number of built-in properties that were tagged in ES3 with the ReadOnly or DontDelete attributes.

Both Mark Miller and Douglas Crockford wanted to add the capability to lock down the properties of an object before passing it to untrusted code. They would use that capability to secure the built-in library objects exposed to a sandbox and allow the code hosting a sandbox to secure any objects that it needed to pass to untrusted code. The repurposing of the DontDelete attribute as the Configurable attribute and the ability to make a property non-Configurable using `Object.defineProperty` provided the basic capability to secure an individual property. But that was insufficient to prevent untrusted code from attaching new properties to an object that was passed to it. The ability to add new properties enabled untrusted code to override inherited behavior and potentially to construct covert communications channels that could be used to leak private data. In ES5 this problem was solved by associating a new state, internally called [[Extensible]], with each object. Objects are initially created with [[Extensible]] set to `true`. But if it gets set to `false` for an object, then new own properties cannot be added to that object. Implementations are forbidden from providing any extensions that can be used to change an object's [[Prototype]] when [[Extensible]] is `false`. Finally, once [[Extensible]] is set to `false` it cannot be reset to `true`.

The function `Object.isExtensible` provides an API for querying an object's [[Extensible]] state. The function `Object.preventExtensions` forces [[Extensible]] to `false`. `Object.freeze` is a convenience function that sets [[Extensible]] to `false` and sets the [[Configurable]] and the

[[Writable]] attributes of all own properties to `false`. This makes the direct state of an object completely immutable. The function `Object.seal` is like `Object.freeze` except that [[Writable]] is not set to `false`. This fixes the prototype and the property set of an object, but still permits the values of data properties to be modified.

Another significant concern was ambient access to the global object. ECMAScript defines the global object as the object whose properties populate the global scope. All of the named standard library objects exist as properties of the global object and most JavaScript hosting environments add additional environment-specific objects and API functions to the global object. For example, in browsers the global object is the same as the `window` object and provides full access to the current Web page's DOM objects and other browser APIs. Typically a sandbox restricts access to some or all of these global object properties or provides alternative versions of some of the global object properties. In theory, it should be possible to accomplish this by imposing an additional lexical scope around all sandboxed code and setting up that scope so it either provides alternative bindings for some global object properties or censors them by providing shadow bindings with the value `undefined`. But since JavaScript 1.0 there has been a way to gain access to the global object that cannot be masked by lexical scoping:

```
function getGlobalObject() {
  //when directly called, the value of this is the global object
  return this;
}
getGlobalObject().document.write("pwned");
```

JavaScript 1.0

Up until ES5, it was specified that the behavior of a direct call (rather than a method call qualified by an object) to a function was to pass `null` as the implicit `this` argument and that all functions, upon entry, would replace a `this` whose value was `null` with the global object. For backward compatibility this could not be changed for existing code. But ES5 strict mode provided an opportunity for new code to opt-in to new behaviors. In ES5, strict mode functions never replace an actual `this` argument with the global object. A sandbox can protect itself from ambient global object access by allowing only strict mode JavaScript code to run in the sandbox.

As ES5 was being developed, actual malicious exploits like the example shown in Figure 35 were starting to be observed on the Web. ES3 specified that objects created using object literals inherit from `Object.prototype` and that object literals use the [[Put]] internal method to install the properties listed in the literal on the new object. But when a value is assigned to an object's property using [[Put]] it looks up the prototype inheritance chain to see if it can find a property with the same name. If it finds a setter property with that name it will execute that setter function. If such a setter is installed on `Object.prototype` then any attempt to use an object literal to create a property with the same name as the setter will call the setter passing it the property value.

Fixing this loophole was a breaking-change to the semantics of object literals, but it is the kind of change that browsers vendors are willing to make to correct security vulnerabilities. The actual specification change was simple: instead of using [[Put]] semantics to create the new object's properties ES5 used the new [[DefineOwnProperty]] internal method which always ignores inherited properties and creates new properties directly on an object.

ES5 could take only small steps toward making JavaScript more secure. While work on ES5 was progressing, Douglas Crockford proposed formation of a Secure ECMAScript (SES) working group within TC39. The purpose [Crockford 2008d; TC39 2008b] was to explore the possibility of developing a secure dialect of ECMAScript that did not have backward compatibility constraints. The SES working group met four times in 2008–2009 and reviewed a number of existing JavaScript

```
// Assume we have discovered that a Web page uses an object literal
// to store some valuable information in a property named "secret"
function setupToStealSecret() {
  // Use non-standard, pre-ES5 getter/setter API
  // to define a getter/setter pair on prototype
  Object.prototype.__defineSetter__("secret", function(val) {
    this.__harmlessSoundingName__ = val; //store value elsewhere
    exploitTheSecret(val, this)
  });
  Object.prototype.__defineGetter__({"secret", function() {
    //get value from alternative location so nothing breaks
    return this.__harmlessSoundingName__;
  });
}
// The secret is leaked whenever the attached code defines an object
// using an object literal with a property named "secret"
var objectWithSecret = {
  secret: "password";  //this triggers the inherited setter.
  //probably defines other properties too
};
```

Fig. 35. Security exploit using JavaScript 1.5's `__defineSetter__` extension. By defining a setter property on `Object.prototype` an attacker can hijack the values of specific properties defined using object literals.

solutions [TC39 2008e] for secure evaluation of untrusted code. Ultimately, the idea of TC39 standardizing a separate new dialect was abandoned but SES concepts such as the object-capability model significantly influenced the Harmony effort. Ankur Taly et al. [2011] formalized how strict mode and other ES5 features enable creation of mashup-friendly secure ECMAScript subsets.

20.1.4 Elimination of Activation Objects. Prior to ES5, the ECMAScript specification had made explicit use of ECMAScript objects to define the scoping semantics of the ECMAScript language. Each *scope contour*[g] was represented by an activation object—an regular ECMAScript object whose properties provided the variable and function bindings created by the code corresponding to that contour. Nested scopes were specified as a list of activation objects which were searched in order for a referenced binding. Language features that referenced bindings used the same property-access semantic operators both for accessing activation objects and for accessing properties of objects defined by the user programs. ES1 and subsequent specifications stated that activation objects were pure specification devices that were unobservable to ECMAScript programs. However, the semantics of property access resulted in several unexpected edge-case behaviors that could be observed if an engine was in full compliance with the specification. Implementations differed in how closely, if at all, they implemented these edge-case semantics.

One anomaly was that activation objects presumably inherit from `Object.prototype` which is the default prototype of newly created objects. That means that the properties of `Object.prototype` are inherited by all activation objects and would appear as local bindings of each activation object shadowing any identically named bindings in outer scopes.

Because binding resolution is specified to occur dynamically, using property lookup on the activation objects, any free reference from within a called function could be intercepted by introducing an `Object.prototype` binding of the referenced name prior to the call, for example:

```
var originalArray = Array;
function AltArray() {
  //this is a replacement for the built-in Array constructor
  //...
}
//Call a function, forcing it to use AltArray in place of Array
Object.prototype.Array = AltArray;
somethingThatFreelyReferencesArray();
delete Object.prototype.Array;  //remove the alternative Array binding
```

ES1–ES3

Another anomaly is ES3 treatment of the parameter of a `try` statement's `catch` clause as a local lexically scoped binding in a new scope which encompasses the body of the `catch` clause. The use of an ECMAScript object to represent scope contours also caused a problem for that semantics. The ES5 specification [Lakshman and Wirfs-Brock 2009, Annex D] describes this problem as follows:

> 12.4: In Edition 3, an object is created, as if by `new Object()` to serve as the scope for resolving the name of the exception parameter passed to a `catch` clause of a `try` statement. If the actual exception object is a function and it is called from within the `catch` clause, the scope object will be passed as the `this` value of the call. The body of the function can then define new properties on its `this` value and those property names become visible identifiers bindings within the scope of the `catch` clause after the function returns. In Edition 5, when an exception parameter is called as a function, `undefined` is passed as the `this` value.

During most of 2008, the working group intended to include `const` declarations in the new edition because it was a feature that was available in 3 out of 4 of the major browsers, although with differing semantics. The plan was to make `const` lexically scoped down to the block level and that was expected to further stress the legacy scoping model used in previous editions of the specification.

To address these issues, Allen Wirfs-Brock developed a new specification-level model of scopes and bindings that did not use ECMAScript object semantics to define the identifier resolution mechanism. The model introduced the concepts of environment records which contain the bindings for a single scope contour and environments which are ordered lists of environment records that provided the context for identifier resolution at a point in an ECMAScript program. There were different kinds of environment records that were used to represent the global scope, function scopes, block scopes, and `with` statement scopes, but all environments expose a common specification-level protocol for the definition, lookup, and value mutation of individual bindings. Language features that declare or access variables and other kinds of bindings are specified using the common environment record protocol.

However, `const` declarations were eventually deferred until a future harmony edition of the specification because the working group realized that early inclusion of `const` might introduce semantics that would encumber a future design for a more comprehensive set of block-scoped declarations. The new scoping model was still used in ES5 to address the known legacy scoping anomalies and provided the foundation for a more comprehensive set of declaration statements in ES6.

20.1.5 Other ES5 Features. In addition to the Object reflection functions listed in Figure 33, ES5 adds the following standard built-in functions, methods, and properties:

- `JSON.parse` and `JSON.stringify` for converting objects from and to JSON interchange format strings

- Nine new `Array.prototype` methods: `indexOf`, `lastIndexOf`, `every`, `some`, `forEach`, `map`, `filter`, and `reduce`, `reduceRight`
- A new `String.prototype` method: `trim`
- `Date`: The `Date.now` method and other extensions to parse and produce data strings in ISO 8601 date format
- A new `Function.prototype` method `bind` and a `name` property on function instances

Other miscellaneous changes and enhancements include:

- Semantic fixes to the scoping of `with` statements and `catch` clause parameters
- Array-like indexing on strings using `[]` syntax
- Minor corrections to the regular expression grammar
- Requiring creation of a new `RegExp` object each time a regular expression literal is evaluated
- Early error reporting of malformed regular expression literals
- The global object properties `undefined`, `NaN` and `Infinity` have read-only values
- Requiring that all specification algorithms use the initial built-in values of `Object`, `Array`, etc. instead of the current value
- Various non-normative semantic clarifications and corrections listed in the Annex D and E of the specification

20.2 Implementations and Tests[86]

During the July 2008 Ecma TC39 meeting in Oslo the committee agreed to have two interoperable implementations of ES3.1 before proceeding to publication. The primary motivation of this "two interoperable implementations" requirement was to ensure that the committee did not standardize something that had not first been proven to be both technically feasible and compatible with existing Web content. Mozilla committed to providing one of the implementations. Because of Microsoft's market position, and historically infrequent browser updates, there were strong feelings within TC39 that Microsoft should demonstrate its commitment to ES3.1 by making publicly available a browser-hosted prototype implementation as part of the ES3.1 validation process. The TC39 plan at that time was to have the ES3.1 ready for publication at the June 2009 Ecma GA meeting. For that to happen a go/no-go decision would need to be made at the March 2009 TC39 meeting based on the result of interoperability testing to be done in the February / March time frame. At the time, there was no official conformance test suite for ECMA-262 and certainly there were no tests of new ES3.1 features. All implementations had their own ad hoc test suites and all of them except for Microsoft also used Mozilla's JavaScript test suite. Microsoft had concerns regarding the Mozilla Public License used for the Mozilla test suite and would not use or contribute to it. Microsoft's preference was for a test suite that would be made available through Ecma using an MIT or BSD style license.

In October 2008, Pratap Lakshman began working on the twin efforts of creating both an Internet Explorer hosted implementation of ES3.1 and a companion test suite.

The test cases that were implemented were to be contributed back to the community. The goal for the test suite was to achieve maximum code coverage, where the "code" was the pseudocode in the specification. Each test case was named after its section and algorithm step numbers in the latest draft of the specification and placed in a single `.js` source file. Figure 36 explains convention used in naming the test files.

Lakshman implemented over 900 test cases and a simple test harness to run and report on the individual testcases. Figure 37 is an example of one of the testcases.

[86]Pratap Lakshman contributed to this section.

`sectionNumber-algorithmStepNumber-testNumber-s.js`

where

sectionNumber:	a section number from the specification
algorithmStepNumber:	the step whose requirements this test validates
testNumber:	optional, present if there are multiple test cases for a particular algorithm step
-s:	optional, present if the test is for strict mode code

Fig. 36. The naming convention used for esconform test case files. Each file contained a single test and the file name encoded the specification pseudocode step that it tested.

```
  // Test Subclause 10.4.2 Alorithm 3 Step 1 Strict mode}
var testName =
"Eval code in strict mode-cannot instantiate variable in calling context";

function testcase() {
  eval("'use strict';var __10_4_2_3_1_s = 1");
  try{
    __10_4_2_3_1_s;
  } catch(e) {
    if (e instanceof ReferenceError)
    return true;
  }
}
```

Fig. 37. An ES5conform Test. This is the test in file `10.4.2-3-1-s.js` from the initial zip file of tests contributed by Microsoft to TC39 [Microsoft 2009a].

At the January 2009 TC39 meeting [Horwat 2009], Pratap Lakshman demonstrated the prototype ES3.1 implementation using an experimental version of `JSCRIPT.dll` integrated into the just-released Microsoft Internet Explorer 8 Release Candidate 1. The demonstration included the new language functionality as well as the conformance test suite. There was general appreciation of this effort, and Waldemar Horwat reported in his meeting notes: "There was much rejoicing among the natives."

Microsoft contributed the tests to Ecma and also released them on its open-source projects portal, codeplex.com, under the name "ES5conform" [2009]. At about the same time, Google announced [Hansen 2009] that it was releasing an open-source ES3 test suite it had created in the course of developing Chrome's V8 JavaScript engine. The test suite, named "Sputnik," consisted of over 5,000 tests.

In 2010, ES5conform and Sputnik became the core of a common Ecma TC39 managed test suite named "Test262." It was a radical change for an Ecma Technical Committee to maintain and distribute a software package, and a number of policy and licensing issues had to be worked out to make this happen. David Fugate led the initial ES5 phase of Test262 development. He was followed by Brian Terlson who got Test262 organized for ES6 and then by Leo Balter in the post ES6 era. Test262 is now an integral part of TC39's development process and tests must be created for every new ECMAScript feature before it is incorporated into the ECMAScript standard. As of August 21, 2018, Test262 consisted of 61,877 tests. The success of Test262 helped convince TC39 that an executable specification was not a necessity.

21 FROM HARMONY TO ECMASCRIPT 2015

The termination of $ES4_2$ provided TC39 with its first opportunity since 1999 to start with a relatively clean slate as it planned the future evolutionary path of JavaScript. No longer was TC39 thinking in terms of starting over to create a better language. TC39 was beginning a path toward success. It would take only seven years to reach its end.

21.1 Getting Started with Harmony

TC39's Harmony project was not constrained by previous decisions made during either of the ES4 efforts but could still draw upon them. TC39 was constrained by decisions made as part of the ES5 project, but that work was now generally in alignment with the direction Harmony was expected to take. In fact much of TC39 meeting time for the latter half of 2008 and most of 2009 was devoted to finishing ES5. This provided an opportunity for the entire committee to grow comfortable with working from an ES5 specification baseline for Harmony.

21.1.1 Strawmen and Goals. In August 2008, the "Harmony Strawman" page was created on the ECMAScript wiki and the es4-discuss mailing list was renamed to es-discuss[g]. Following the Harmony announcement there was an eruption of new discussions on es-discuss about potential Harmony features. The workflow that developed was that new ideas would surface either on es-discuss or at TC39 meetings. If a TC39 member thought an idea had merit, they would write up a preliminary design or description of the feature and post it on the Strawman wiki page. The strawman would be presented at a TC39 meeting, and depending upon the committee's reaction, the idea would either be abandoned or the process would be iterated to refine the idea. On November 21, 2008, the wiki Strawman page [TC39 Harmony 2008] listed the following entries:

- classes
- const
- lambdas
- lexical scope
- names
- return to label
- types

All the entries pointed to short strawman proposals created by Dave Herman except for classes which was a placeholder.

The discussions of possible Harmony features were expansive, and by the summer of 2009 the committee decided to impose more structure on the effort. At the July 2009 meeting [TC39 2009a] the TC39 members decided it was time to define the goals for Harmony. They concluded that the ES3.1 goals [Crockford 2008a] were still applicable with a few additions and refinements. Brendan Eich [2009a] posted an updated version of those goals. The result is the Harmony Goals Statement shown in Figure 38.

21.1.2 The Champions Model. Dave Herman proposed to the committee that it should adopt a "champions model" of development.[87] Applying the champions model, an individual member or collectively a small group of members are responsible for an individual feature. The champion writes an initial strawman proposal and attempts to refine it until it is ready to be integrated into the actual specification. Starting with the initial strawman and subsequently as the proposal evolves, the champion makes presentations to the entire committee and accepts feedback from the committee and other reviewers. It is up to the champion to digest the feedback and to make decisions regarding whether to update the proposal based upon the feedback. According to the

[87] Herman does not recall where he encountered this idea, but it is likely related to Oscar Nierstrasz's [2000] Champions Patterns for organizing program committees.

Requirements

1. New features require concrete demonstrations.
2. Keep the language pleasant for casual developers.
3. Preserve the "start small and iteratively prototype" nature of the language.

Goals

1. Be a better language for writing:
   I. complex applications;
   II. libraries (possibly including the DOM) shared by those applications;
   III. code generators targeting the new edition.
2. Switch to a testable specification, ideally a definitional interpreter hosted mostly in ES5.
3. Improve interoperation, adopting de facto standards where possible.
4. Keep versioning as simple and linear as possible.
5. Support a statically verifiable, object-capability secure subset.

Means

1. Minimize the additional semantic state needed beyond ES5.
2. Provide syntactic conveniences for:
   I. good abstraction patterns;
   II. high integrity patterns;
   III. defined by desugaring into kernel semantics.
3. Remove (via opt-in versioning or pragmas) confusing or troublesome constructs.
   I. Consider making Harmony build on ES5 strict mode.
4. Support virtualizability, allowing for host object emulation.

Fig. 38. Harmony Goals Statement, July 2009 [Eich 2009a]

champions model, the committee should avoid falling back into design by committee behavior during champion presentations. Ultimately, it still takes a consensus decision of the full committee to include a final proposal in the specification.

The committee accepted Herman's proposal to follow the champions model and generally used it effectively. However, there were times when it broke down. The core group of members during this period was relatively small and very technically competent. Sometimes they simply could not resist the temptation to do a little bit of design by committee and sometimes this was actually the most effective way to make progress on a proposal. At times multiple champions emerged with alternative approaches and proposals for a particular feature or design problem. In those cases, if the competing champions could not come to agreement on a common proposal, the committee would have to choose one or in some cases reject all of the competing proposals.

21.1.3 Choosing a Feature Set. For most of 2009, 2010, and the first half of 2011, TC39 champions worked on developing strawman proposals, vetted them with the committee, and tried to get the consensus needed to advance to accepted proposal status. By August of 2009, the Strawman page [TC39 Harmony 2009] had grown to 21 proposals from its original 7. By early 2010, the general shape of the Harmony feature set was beginning to emerge. Brendan Eich [2010a] organized them into a set of themes (Figure 39) that he added to the Harmony Goals page. By December 2010, the Strawman page [TC39 Harmony 2010b] had grown to 66 proposals, and an additional 17 proposals [TC39 Harmony 2010a] had already been deferred or abandoned. By the beginning of

Themes

1. Modularity, or how to delineate units of source code to hide the insides from outside users
2. Isolation, to prevent effects from propagating, or allow them only through certain references
   - Zero-authority maker-style modules
   - Other combinations of primordials/contexts/builtins with modules
   - Lack of isolation in browsers: multiple connected global objects
3. Virtualization, for stratified guest code hosting, bridging disjoint object systems, and in particular emulating host objects
   - Proxies
   - Weak references or Ephemerons
4. Control Effects, for simpler iteration and state-machine code
   - Delimited continuations
   - Generators, iterators
5. Library/Tool Enabling, so the TC39 committee is not blocking library evolution
   - Object.hashcode
   - Byte arrays of some kind
   - Value types (for Decimal, etc.)
6. Language Reform, the "better carrots" needed to lead users away from bad forms
   - let, const, function in block scope
   - Default and rest parameters, the spread operator
   - Destructuring
7. Versioning, since new syntax is part of Harmony
   - This theme is about minimizing opt-in versioning, easing migration, and future-proofing for the next edition

Fig. 39. 2010 Harmony Feature Themes [Eich 2010a]

May 2011 the Strawman page [TC39 Harmony 2011c; Appendix N] had over 100 entries and the approved Proposals page [TC39 Harmony 2011a] had 17 entries.

In 2009, Brendan Eich [TC39 2009b] had proposed that TC39 target June 2012 for Ecma GA approval of "ES.next" with a feature freeze target of May 2011. As the May target date approached it was obvious that June 2012 was not achievable, but it still made sense to draft a committed feature list to focus specification development. The majority of the May meeting [TC39 2011b] was devoted to triaging the strawman list and reaching consensus on which remaining strawman proposals would advance to "Harmony Proposal" status. Each strawman proposal was discussed prior to gauging whether or not there was consensus to advance it. Some proposals were advanced or rejected after minimal review. Other proposals representing important functionality were advanced even though the committee was not satisfied with the then-current strawman. Those proposals served as placeholders, pending development of improved proposals. Both modules and classes were given that treatment. The final Harmony feature set was not strictly frozen at that meeting. As development of ES.next continued, some proposals were added and some were dropped. But the proposal list from this meeting established the general shape of what would become ES2015. Figure 40 lists the participants at the May meeting, and Appendix O shows the Harmony Proposal page [TC39 Harmony 2011b] following the meeting.

21.1.4 Writing Starts. Allen Wirfs-Brock, as project editor, had overall responsibility for creating the ES.next specification document from the Harmony proposals developed by TC39 champions. At

Avner Aharon	Microsoft	Waldemar Horwat	Google
Douglas Crockford	Yahoo! (Phone)	Mark Miller	Google
Brendan Eich	Mozilla	John Neumann	Ecma
Cormac Flanagan	UCSC	Alex Russell	Google
David Fugate	Microsoft	Mike Samuel	Google
Dave Herman	Mozilla	István Sebestyén	Ecma
Luke Hoban	Microsoft	Sam Tobin-Hochstadt	Northeastern Univ
Bill Frants	Periwinkle (guest)	Allen Wirfs-Brock	Mozilla

Fig. 40. Attendees at May 2011 TC39 Feature Winnowing [TC39 2011b]

5.1.4	Introduction of the concept of supplemental grammars.
5.3	Introduction of concept of Static Semantic Rules.
8.6.2	[8.6.2] and various places. Eliminated [[Class]] internal property. Added various internal trademark properties as a replacement.
10.1.2	Defined the concept of "extended code" that means code that may use new Es.next syntax. Also redefined "strict code" to mean either ES5 strict mode code or extended code.
11.1.4	Added syntax and semantics for use of spread operator in array literals.
11.1.5	Added syntax and semantics for property value shorthand and various semantic helper abstract operations.
11.2, 11.2.4	Added syntax and semantics for spread operator in argument lists
11.13	Add syntax and semantics for destructuring assignment operator.
12.2	Added BindingPattern syntax and partial semantics to support destructuring in declarations and formal parameter lists.
13	Added syntax to support rest parameter, parameter default values, and destructuring patterns in formal parameter lists. Also static semantics for them. However, instantiation of such parameters is not yet done. Defined the argument list "length" for such enhanced parameter lists.
15	Clarified that clause 15 function specifications are in effect the definition of [[Call]] internal methods.
15.2.4.2	Respecified toString to not use [[Class]]. Note that adding an explicit extension mechanism is still a to-do.
Annex B	Retitled as normative optional features of Web Browser ES implementations.

Fig. 41. Change log for 1st ES6 Draft [Wirfs-Brock et al. 2011a, reformatted]

Microsoft his responsibilities were split between TC39 related work and other projects. In December 2010, he left Microsoft and went to work for Mozilla to focus on ES Harmony.

The ES4 and ES5 experiences had taught Wirfs-Brock that continual work on the actual specification document was essential to making progress toward completing a new edition of the standard. On June 22, 2011, with stern determination he opened the source file for the recently completed ES5.1 specification, changed the cover page to say "Draft, Edition 6," and saved it as the baseline ES6 draft specification. He then immediately started editing new material into that draft based on the May feature triage and other decisions that had been made by the committee over the previous two years. On July 12, he posted the "first working draft of the ES.next specification" [Wirfs-Brock et al. 2011a,b]. Figure 41 is the change summary for that draft. It was the first of 38 distributed drafts, the last of which was posted to the wiki on April 14, 2015 [Wirfs-Brock et al. 2015a,c].

21.1.5 One JavaScript. From the beginning of the Harmony effort, TC39 assumed that some sort of explicit opt-in would be required in order to use many or perhaps all new Harmony features. This

was a carry-over from the ES4 era where a number of proposals included breaking-changes that would invalidate some existing JavaScript programs. The Harmony process was more conservative about including breaking-changes, but a few were still contemplated. Over the first three years of Harmony development, the exact opt-in mechanism was undecided but frequently discussed. The first ES6 draft incorporated the concept of "extended code" which was a superset of ES5 strict code but did not yet include a description of the opt-in mechanism. Alternatives considered included external opt-in using an attribute of the HTML `<script>` element, a new `use mode` pragma statement, a delimited syntactic form, or adding an additional directive similar to `"use strict";`. There was concern about how many modes there would be in the future. Was each major edition of the standard going to require a new opt-in mode? This seemed like a major complexity burden for both users of the language and implementors.

Dave Herman [2011b], in an es-discuss message titled "ES6 doesn't need opt-in," argued that breaking-changes should be very limited and restricted to code that is encapsulated as an ES6 module. The vast majority of features should be non-breaking and behave identically whether or not they occur within modules. In some cases that might require redesigning some features and in a few cases contemplated features might have to be dropped. These ideas were refined in the course of over 150 replies to the es-discuss message. At the next TC39 meeting, Herman [2012] made a presentation titled "One JavaScript," which introduced a refinement of these ideas. The key point was that future programmers and implementors of ECMAScript Harmony should be able to think in terms of one unified JavaScript language without thinking about modes, versions, or dialects. It was the responsibility of TC39 to design ES.next to be consistent with that perspective. Much of the meeting was devoted to discussion of this proposition and the impact it would have on various Harmony features. The consensus was to try to make "1JS" work for Harmony. In the next specification draft [Wirfs-Brock et al. 2012a], the concept of extended code was gone and various other changes were made to eliminate what would have been breaking-changes.

21.1.6 Brendan's Dreams. In January 2011, after more than two years of work on Harmony, Brendan Eich [2011b] published a blog post titled "Harmony of My Dreams," in which he presented some opinions about language evolution and standards committees. The heart of the post provides examples of what he hoped Harmony JavaScript would be like:

> …I'd like to present a refreshed vision of JavaScript Harmony. This impressionist exercise is of course not canonical (not yet), but it's not some random, creepy fanfic either. Something like this could actually happen, likelier and better if done with your help (more on how at the end).
>
> I'm blurring the boundaries between Ecma TC39's current consensus–Harmony, straw proposals for Harmony that some on TC39 favor, and my ideas. On purpose, because I think JS needs some new conceptual integrity. It does not need play-it-safe design-by-committee, either of the "let's union all proposals" kind (which won't fly on TC39), or a blind "let's intersect proposals and if the empty set remains, so be it" approach (which also won't fly, but it's the likelier bad outcome).

He presents examples of how various use cases are coded using ES5 features and alternative examples of how the same things could be expressed in the Harmony of his dreams. The dream examples provide an intermediate stage view of Harmony proposals and how they evolved into actual ES2015 features. Some of what he presented was not included in ES2015 and most of the features ultimately changed in some ways. Other changes were necessary because the 1JS approach eliminated the possibility of opt-in changes to the syntax and semantics of existing features.

To get a perspective on this feature evolution, compare some of Brendan Eich's 2011 dreams[88] with what eventually became the reality of ES2015.

Dream—Binding and Scoping. Block scoped declarations and free variable references are early (parse-time) errors:

```
let block_scoped = "yay!"
const REALLY = "srsly"
function later(f, t, type) {
  setTimeout(f, t, typo) // EARLY ERROR
}
```

ES2015 reality: Block-scoped `let` and `const` declarations but 1JS precludes early errors for free variable references.

Dream—Improved Function Definitions. Eliminate `function` keyword, implicit `return` of last expression, eliminate redundant closures for functions with no free variables:

```
const #add(a, b) { a + b }
#(x) { x * x }
```

ES2015 reality: Arrow functions replaced the # notation. Implicit return only for arrow functions with expression bodies. Concise methods in object literals and class bodies. Unobservable closure optimizations left to implementations:

```
const add = (a, b) => a + b //expression body has implicit return
x => x * x
x => {console.log(x); return x * x} //statements body needs explicit return
// method definition in object literals and classes
class {
  add(a, b) {return a + b}  // No expression bodies
}
```

Dream—Lexical this. In hash functions `this` is lexically bound to enclosing `this` binding:

```
function writeNodes() {
  this.nodes.forEach(#(node) {
    this.write(node)
  })
}
```

ES2015 reality: `this` and other function-scoped implicit bindings lexically bound in arrow functions.

```
function writeNodes() {
  this.nodes.forEach(node => this.write(node))
}
```

Dream—records and tuples. Immutable data structures with content based equality:

```
const point = #{x: 10, y: 20}
point === #{x: 10, y: 20}  //true
```

ES2015 reality: Not included. Too closely linked to the concept of extensible value types which did not get fully developed for Harmony.

[88]The dream labels and descriptions are paraphrased from Brendan Eich's blog post. The dream code snippets are direct quotes. Notice that he chose to use a semicolon free style when coding the dreams.

Dream—Rest, spread, and destructuring. Syntactic support for variable length argument lists, expanding arrays into argument lists and array literals, and extracting components from arrays and objects.

```
function printf(format, ...args) {
  /* use args as a real array here */
}
function construct(f, a) {
  return new f(...a)
}
let [first, second] = sequence
const {name, address, ...misc} = person
```

ES2015 reality: Exactly the same except that in ES2015 did not support object destructuring using the ... operator. That feature was added to a subsequent edition.

Dream—modules. A simple modularity design that supports asynchronous loading in browsers.

```
module M {
  module N = "http://N.com/N.js"
  export const K = N.K  //value of N.K exported
  export #add(x, y) { x + y }
}
```

ES2015 reality: One module per file with no explicit module definition delimiters. More `import` and `export` formulations. Bindings rather than values shared among modules.

```
// content of http://M.com/M.js
export {K} from "http://N.com/N.js" //binding of N.K exported
export const add = (x, y) => x + y
```

Dream—iteration. Paren-free `for-in` statement extended to work with iterators provided by proxy-based standard library or user defined generator functions:

```
module Iter = {"@std:Iteration"}
import Iter.{keys,values,items,range}
for k in keys(o) { append(o[k]) }
for v in values(o) { append(v) }
for [k,v] in items(o) { append(k, v) }
for x in o { append(x) }
#sqgen(n) { for i in range(n) yield i*i }
return [i * i for i in range(n)]  //array comprehension
return (i * i for i in range(n))  //generator comprehension
```

ES2015 reality: 1JS motivated the `for-of` statement as the alternative to overloading `for-in` via dependency on modules and proxies. Standard key/value/entries protocol defined for built-in collection classes. Comprehensions were dropped late in Harmony development because of future-proofing concerns.

```
for (k of o.keys()) append(o[k])
for (v of o.values()) append(v)
for ([k,v] of o.entries()) append(k, v)
for (x of o) append(x) // o provides its default iterator
function *sqgen(n) {for (let i of Array(n).keys) yield i*i }// a generator
```

Dream—paren-free statements. More modern syntax that eliminates required parenthesizes in compound statements:

```
if x > y { alert("paren-free") }
if x > z return "brace-free"
if x > y { f() } else if x > z { g() }
```

ES2015 reality: Not included. Rejected by TC39 as too radical. 1JS required continued recognition of old forms and mixing of old and new forms resulted in extra design and user complexity.

21.2 Recrafting the Specification

The desire to express the ECMAScript semantics using an executable, testable specification had carried forward from the $ES4_2$ effort, but ML had been discarded as a specification language. Early in the Harmony effort Allen Wirfs-Brock [2009] had floated the idea of using a definitional interpreter written in ES5 JavaScript to specify Harmony. That idea had even been included in the Harmony Goals Statement (Figure 38). But by the spring of 2010 not much progress had been made on that concept, and TC39 members were less sure of that approach. The pseudocode improvements made for ES5 (Appendix Q) had eliminated most of usability issues that had existed with the pseudocode in earlier editions. Progress with Test262 showed that a comprehensive test suite was useful for verifying the specification as well as implementations. The specification formalism was again discussed at the May TC39 [2010] meeting, and the status quo remained attractive to many at the meeting. Apple's Oliver Hunt observed that, as an implementor, the pseudocode in ES5 worked better for him than any executable specification code he had seen. The consensus decision was to continue to use pseudocode to specify Harmony.

For the project editor, creating the specification was more than a simple integration task. In theory, proposals were developed by champions to the point where they were ready for easy integration into the specification. In practice, this was rarely the case. Some champions were not familiar enough with the structure of the specification or its formalisms to create integration-ready pseudocode. Others did not have the time or expertise necessary to create detailed semantics specifications. For many proposals, Allen Wirfs-Brock had to devise how to integrate them into the specification, work out semantic details, and write or rewrite the proposal's specification algorithms.

Champions tended to have a narrow focus on the features defined by their proposals. Good proposals take into consideration how the feature interacts with preëxisting features of the language. However, even the most skilled champions have a hard time considering all the potential interactions between their features and other proposals which are being simultaneously developed by other champions. All the features had to pass through the editor in order to become part of the actual specification, so Wirfs-Brock had the most complete view of how the existing language and all the Harmony proposals would fit together to form ES6. He was particularly focused on cross-cutting concerns that span multiple feature proposals, and ensuring that there was syntactic and semantics consistency among proposals. As he integrated approved proposals he tried to transform them into a set of composable orthogonal features [Lindsey 1993]. Sometimes this required changing syntactic or semantic details of a proposal or even adding or removing significant functionality. These changes would then have to be presented to the champions and often the full committee for approval.

21.2.1 Reorganizing the Specification. From the first draft of the first edition in 1997 (Figure 13) through ES5.1, the organization of the ECMAScript specification had remained fundamentally unchanged. While working on the ES5 specification, Allen Wirfs-Brock had found the basic ordering

Clause	ECMA-262, 5.1 Edition (245 pages)	ECMA-262, 6th Edition (545 pages)
1	Scope	Scope
2	Conformance	Conformance
3	Normative References	Normative References
4	Overview	Overview
5	Conventions	Notational Conventions
6	Source Text	ECMAScript Data Types and Values
7	Lexical Conventions	Abstract Operations
8	Types	Executable Code and Execution Contexts
9	Type Conversion and Testing	Ordinary and Exotic Object Behaviors
10	Executable Code and Execution Contexts	ECMAScript Language: Source Code
11	Expressions	ECMAScript Language: Lexical Grammar
12	Statements	ECMAScript Language: Expressions
13	Function Definition	ECMAScript Language: Statements and Declarations
14	Program	ECMAScript Language: Functions and Classes
15	Standard Built-in ECMAScript Objects	ECMAScript: Language: Scripts and Modules
16	Errors	Error Handling and Language Extensions
17		ECMAScript Standard Built-in Objects
18		The Global Object
19		Fundamental Objects
20		Numbers and Dates
21		Text Processing
22		Indexed Collections
23		Keyed Collections
24		Structured Data
25		Control Abstraction Objects
26		Reflection

Fig. 42. Organization of 5th and 6th Editions. In the ES6 specification clauses 6–9 define the virtual machine semantics. Clauses 10–15 define the language and clauses 17–26 define the standard library.

of the material in the specification confusing. He came to understand that the specification defined three fundamentally separable parts:

- an ECMAScript virtual machine including its runtime entities and their semantics
- the ECMAScript language syntax, semantics, and its mapping to the virtual machine
- the standard library of objects that is available to all ECMAScript programs

The original specification and its revisions interwove the three parts in a manner that obscured this basic structure. Allen Wirfs-Brock felt that explicitly organizing the specification into a three-part structure would make it easier to understand and would provide a clearer presentation of the large amount of new ES6 material. The committee agreed. Figure 42 shows the new organization of the ES2015 specification compared with that of ES5.

21.2.2 New Terminology. ES6 provided an opportunity to clarify and update some of the terminology used in the specification. One area that needed attention was nomenclature for objects. JavaScript 1.0 implementations had given JavaScript programs access to host-specific and JavaScript engine-specific objects whose fundamental object semantics differed in various unusual ways from the objects that could be created using ECMAScript code. The ES1 specification used the terms "object," "native object," "standard object," "built-in object," "standard native object," "built-in native object," and "host object" to refer to the various ways objects could be implemented. The distinctions between these designations were subtle and not particularly useful. It was not clear exactly which

of these categories permitted unusual object semantics or whether objects created by JavaScript programmers fit into any of the categories.

A goal of ES6 was to enable self-hosted implementations of most standard library and host objects using JavaScript code. With the possibility of self-hosting, the distinction between host-provided, engine-provided, and program-provided had diminishing relevance. The semantic differences among objects are more important than who provides them or the technology used to implement them.

The fundamental terminology need was to distinguish objects with normal semantics from those with abnormal (that is, unusual) semantics. Douglas Crockford [TC39 2012b], riffing on the name of Ecma's highest membership category, suggested "ordinary object[g]" as the term for objects that have the semantics of an object created using a JavaScript object literal or `new Object()`. Objects whose semantics deviated in any way from the ordinary object semantics were called "exotic objects[g]." Both ordinary and exotic objects might be provided by the host, engine, or application programmer, and might be implemented using JavaScript or some other language.

21.2.3 New Kinds of Semantics. Prior to ES6, the majority of pseudocode algorithms, other than those that defined standard library functions, were associated with grammar productions and specified the runtime evaluation semantics of their production. There was no need to name these algorithms because they were the only semantics associated with the grammar productions. There were also a few algorithms, such as those for type conversions and the internal methods that define object semantics, which were not directly associated with the grammar. Those were given names so that they could be referenced from the evaluation algorithms.

ES6 introduced new features, such as object destructuring, which have complex behaviors whose specifications must cross-cut many grammar productions. Some of the algorithms need to perform multiple traversals of the parse tree to collect information or to sequence evaluation steps that span multiple parse nodes. There are also common grammar-linked behaviors that, for consistency, are used by multiple features. To accommodate these requirements, the ES6 specification can associate named algorithms with parse nodes in addition to the implicitly named evaluation algorithms. They are referenced by name in association with grammar symbols. Typically such named algorithms are polymorphic in the sense that a same-named algorithm is defined for multiple grammar productions. The specific algorithm selected depends upon the actual derivation of the grammar symbol used to parse a specific source text.

With the goal of minimizing implementation variation, each subsequent edition of ECMA-262 was more precise in its definition of error conditions and when they are detected. ES3 implicitly introduced the concept of "early errors" which was further refined in ES5. An early error is an error in a script that is detected and reported prior to evaluation of the script. Detection of an early error prevents evaluation of the script. The most common form of early error is a syntax error which occurs when the source code of a script cannot be parsed using the ECMAScript grammar. Syntax errors are implicit in the definition of the grammar. ES3 introduced a few other kinds of early errors, for example a `break` statement that references a statement label that does not lexically enclose the `break` statement. ES5 strict mode added a few more. The specification defined most of these as syntax errors even though they are not parsing errors but instead are violations of the static semantic rules of the language. Prior to ES6, most such errors were specified using informal prose placed near an evaluation algorithm. Others were specified by including pseudocode that tested for runtime error conditions within evaluation algorithms and then used prose to state that the error could or should be reported as an early error.

ES6 features introduced many more kinds of early errors. For example, it is an early error to attempt to multiply define an identifier using `let` or `const` declarations. ES6 added "Static Semantics"

13.3.1.1 Static Semantics: Early Errors

LexicalDeclaration : *LetOrConst BindingList* **;**

- It is a Syntax Error if the BoundNames of *BindingList* contains `"let"`.
- It is a Syntax Error if the BoundNames of *BindingList* contains any duplicate entries.

LexicalBinding : *BindingIdentifier Initializer*$_{\text{opt}}$

- It is a Syntax Error if *Initializer* is not present and IsConstantDeclaration of the *LexicalDeclaration* containing this production is **true**.

...

13.3.1.3 **Static Semantics: IsConstantDeclaration**

LexicalDeclaration : *LetOrConst BindingList* **;**

1. Return IsConstantDeclaration of *LetOrConst*.

LetOrConst : **`let`**

1. Return **false**.

LetOrConst : **`const`**

1. Return **true**.

Fig. 43. Sample ES6 Static Semantic Rules [Wirfs-Brock 2015a, pages 194–195]

subclauses to the grammar for consistently specifying early error conditions. Figure 43 shows a sample set of early error definitions. As shown, early-error rules may reference static semantic algorithms. Static semantic algorithms use the same conventions as the runtime algorithms except they may not reference any of the runtime state of the ECMAScript environment—because they are applied before evaluation of a script. Static semantic early-error rules and algorithms are restricted to using and analyzing information that can be extracted from source code without executing it. Runtime algorithms may invoke static semantic algorithms but static semantic algorithms may not invoke runtime algorithms.

21.3 ES2015 Language Features

The proposals listed in the final versions of the Harmony Proposals wiki page [TC39 Harmony 2014] were developed into dozens of new and extended language and standard library features. Typical proposals went through multiple iterations before being incorporated in the draft specification, and in some cases further evolved after that. A number of proposals were eventually dropped from consideration or deferred to future editions.

The following sections take a deeper look at the development history of a few key proposals and summarizes details of other important features.

21.3.1 Realms, Jobs, Proxies, and a MOP. The Harmony goals included enabling the self-hosting of built-in and host-defined exotic objects and fully specifying the semantic extensions that were implemented by Web browsers. To support this goal, it was necessary to refine some of the existing abstractions of the ECMAScript "virtual machine" and to add new abstractions that could be used to specify language features that were new or underspecified.

"Realm" [Wirfs-Brock 2015a, pg. 72] is a new specification abstraction added to describe the semantics of multiple global namespaces within a single ECMAScript execution environment. Realms support the semantics of HTML frames (§3.6), a feature of browsers that had been ignored by ECMAScript since ES1. "Job" [Wirfs-Brock 2015a, pg. 76] is a specification abstraction added to deterministically define how an ECMAScript execution environment sequentially executes multiple

scripts, each of which runs to completion. Jobs provide a way to explain the semantics of event dispatching and deferred callbacks provided by browsers and other JavaScript hosts. They also provide the basis for defining the semantics of ES2015 Promises.

The internal methods provided by ES1 (§9) were essentially a vestigial metaobject protocol. The intent was that variations in the observable semantics of property access among various kinds of built-in and host-provided objects could be explained as differences in the specification of their internal methods. But prior to ES2015, the internal methods semantics was incomplete, underspecified, and used inconsistently. In order to tame host objects, enable self-hosting of exotic objects, and support object-capability *membranes*[g] [Van Cutsem and Miller 2013], the ES1–ES5 internal method design was transformed into a fully specified MOP.

For JavaScript code to define exotic objects, it has to be able to provide the implementation of the internal methods of those objects. This capability is provided by ES2015 `Proxy` objects [Wirfs-Brock 2015a, pg. 495]. $ES4_2$ proposed a mechanism called "catchalls" [TC39 ES4 2006a] which was intended to enable JavaScript code to override, on a per object basis, the default actions that occur when attempting to access a property or invoke a method that does not exist. $ES4_2$ catchalls were intended to be an improvement on JavaScript 1.5's non-standard `__noSuchMethod__` mechanism [Mozilla 2008a]. For Harmony, Brendan Eich [2009b; 2009d] generalized $ES4_2$ catchalls by introducing the concept of dynamically attaching what he called "action methods" to an object. Performing certain language operations on an object would call an action method if one was defined for that object. The set of available actions were similar to the set of ES5 internal methods but not a direct reflection of them. An open issue was whether the actions would be triggered on all property access or on accesses only to non-existent properties. Eich's API for attaching actions to an object was modeled on the ES5 Object Reflection functions:

```
var peer = new Object;
Object.defineCatchAll(obj, {
  // add action methods that implement Array-like length behavior
  has: function (id) { return peer.hasOwnProperty(id); },
  get: function (id) { return peer[id]; },
  set: function (id, value) {
       if ((id >>> 0) === id && id >= peer.length) peer.length = 1 + id;
       peer[id] = value},
  add: function (id) { Object.defineProperty(obj, id,
        { get: function (){ return peer[id]; },
        set: function (value) { peer[id] = value);
       }})},
  // definitions of other actions ...
});
```

Harmony Catchall Proposal

In this example, the properties `has`, `get`, `set`, and `add` provide the catchall actions that get dynamically attached to object `obj`. The action functions lexically share access to the `peer` object. This establishes a one-to-one association between `obj` and `peer`. The handlers work together to use `peer` as the backing store for what will appear to be own properties of `obj`. They also dynamically update the value of the `peer` object's `length` property so its value is always one greater than the largest integer used as a property name.

Brendan Eich's catchall proposal was shortly followed by an alternative design championed by Tom Van Cutsem and Mark Miller [2010a; 2010b]. Announced [Van Cutsem 2009] as "Catch-all proposal based on proxies," it defined a stratified object-intercession API. The intent of the Proxy proposal was to enable definition of virtual objects, such as the membrane objects used for isolation

in secure object-capability–based systems. TC39 was generally receptive to the Proxy strawman and quickly accepted it as a Harmony proposal.

The proposal introduced the concept of Proxy objects. Instead of extending a base object with interceding action methods, a Proxy object is created with an associated handler object whose methods are called "traps." Traps are triggered by language operations. A handler could completely define the object behaviors used by language operations. The traps might be self-contained or they might work in conjunction with preëxisting objects known to the handler via lexical capture, for example [Van Cutsem and Miller 2010c]:

```
// a simple forwarding proxy
function makeHandler(obj) { return {
  has: function(name) { return name in obj; },
  get: function(rcvr,name) { return obj[name]; },
  set: function(rcvr,name,val) { obj[name]=val; return true; },
  enumerate: function() {
    var res = []; for (name in obj) { res.push(name); }; return res; },
  delete: function(name) { return delete obj[name]; } };
}
var proxy = Proxy.create(makeHandler(o), Object.getPrototypeOf(o));
```

Initial Harmony Proxy Proposal

In this example, `makeHandler` is a helper function that is used to create a handler object whose traps lexically share access to the object that was passed as an argument to `makeHandler`. The object passed to `makeHandler` might be a newly created object, in which case it can serve a role similar to the `peer` object in the catch-all example. Alternatively, the passed object could be a preëxisting object. In that case, the traps could forward some or all of their trapped operations to that object. In that case the object serves the role of the target of a "forwarding proxy."

Placing the trap methods in the handler object avoids name conflicts with base-object properties. The proposal defined seven fundamental traps, six derived traps,[89] and two traps which are specific to function objects. As with the catchall proposal, the traps were similar to the ES5 internal methods but not a direct reflection of them. ES5 established certain invariants [Wirfs-Brock 2011b, page 33], which must not be violated, for the [[GetOwnProperty]] and [[DefineOwnProperty]] internal methods. A troublesome issue for ES2015 was how to virtualize frozen/sealed objects and non-configurable properties while enforcing[90] those invariants.

Prototyping the original Proxy proposal led to a major revision announced by Van Cutsem [2011]:

> A couple of weeks ago, Mark and I sat together to work on a number of open issues with proxies, in particular, how to make proxies work better with non-configurable properties and non-extensible objects. The result is what we call "direct proxies": in our new proposal, a proxy is always a wrapper for another "target" object. By slightly shifting our perspective on proxies in this way, many of the earlier open issues go away, and the overhead of proxies may be substantially reduced in some cases.

In the Direct Proxies proposal [Van Cutsem and Miller 2011a,b, 2012], the target object (`o` in the following example) is like the object passed to `makeHandler` in the forwarding Proxy example. It is kept as internal state of the Proxy object and is passed as an explicit argument when a trap is called. Because the Proxy knows the target object it can use the target to enforce the essential invariants. The following is a Direct Proxies version of the forwarding Proxy example:

[89] Fundamental traps are primitives. The default behaviors of the derived traps are defined using the fundamental traps.
[90] ES5 did not support user-defined internal methods, so explicit enforcement of the invariants was unnecessary.

```
// a simple direct forwarding proxy
var Proxy(o,{
  //the handler object
  has:function(target,name){return Reflect.has(target,name)},
  get:function(target,name,rcvr){return Reflect.get(target,name,rcvr)},
  set:function(target,name,val,rcvr){return Reflect.set(target,name,val,rcvr)},
  enumerate:function(target){return Reflect.enumerate(target)},
  // ...
});
```

Harmony Direct Proxy Proposal

The methods of the `Reflect` object correspond to the standard internal methods. They enable a handler to directly invoke an object's internal methods rather than using JavaScript code sequences that implicitly invoke them. The Direct Proxy design initially defined sixteen different traps largely based upon the ES5 internal methods. The design also identified a number of internal operations on objects that could not be intercepted by a Proxy because they were not defined in terms of internal methods. Tom Van Cutsem, Mark Miller, and Allen Wirfs-Brock worked together to co-evolve the Harmony internal methods and the Proxy traps so that they were aligned and adequate to express all of the object behaviors defined by the ECMAScript specification and by host objects. This was accomplished by adding new internal methods and by redefining some non-interceptable operations as regular base-level, trappable method calls. Essential invariants for each internal method were defined. ECMAScript implementations and hosts are required to adhere to those invariants and `Proxy` enforces[91] them for self-hosted exotic objects. Figure 44 is a summary of the ES2015 MOP.

The Direct Proxy design uses an encapsulated target object, but it is not intended to provide easy transparent wrapping of the target object. Contrary to appearances, Proxies are not a simple way to

ES5 Internal Method	**ES6 Internal Methods**	**ES6 Proxy Traps & Reflect methods**
[[CanPut]]		
[[DefaultValue]]		
[[GetProperty]]		
[[HasProperty]]	[[HasProperty]]	has
[[Get]]	[[Get]]	get
[[GetOwnProperty]]	[[GetOwnProperty]]	getOwnPropertyDescriptor
[[Put]]	[[Set]]	set
[[Delete]]	[[Delete]]	deleteProperty
[[DefineOwnProperty]]	[[DefineOwnProperty]]	defineProperty
[[Call]]	[[Call]]	apply
[[Construct]]	[[Construct]]	construct
	[[Enumerate]]	enumerate
	[[OwnPropertyKeys]]	ownKeys
	[[GetPrototypeOf]]	getPrototypeOf
	[[SetPrototypeOf]]	setPrototypeOf
	[[IsExtensible]]	isExtensible
	[[PreventExtensions]]	preventExtensions

Fig. 44. The ES6/ES2015 Metaobject Protocol is defined by the specification-level internal methods and reified via `Proxy` traps and `Reflect` methods.

[91] A design concern was the cost of enforcing the invariants after each Proxy trap. Notification Proxies [Van Cutsem 2013] is a alternative design that was briefly considered as a way to eliminate that overhead.

log property access or handle "method not found". Naïvely implemented Proxy objects, intended to support such use cases, are often unreliable or buggy. The core use cases for Direct Proxies are the virtualization of objects and the creation of secure membranes. As Mark Miller [2018] explained:

> Proxies and WeakMaps were designed, and initially motivated, to support the creation of membranes. Proxies used standalone cannot be transparent, and cannot reasonably approximate transparency. Membranes come reasonably close to transparently emulating a realm boundary. For classes with private members, the emulation is essentially perfect.

21.3.2 Block-Scoped Declarations. Adding block-scoped lexical declarations was contemplated beginning with the first $ES4_1$ attempt. Programmers experienced with C-like language syntax expect declarations located within a `{}` delimited block to be local to that block. The original JavaScript 1.0 `var` scoping rules (§3.7.1) are surprising and sometimes mask serious bugs. One such bug that is commonly encountered is the closure-in-loop bug:

```
function f(x) {  //this function has the closure in loop bug
  for (var p in x) {
    var v = doSomething(x, p);
    obj.setCallback(function(arg) {handle(v, p, arg)});
    //bug: all the closures created in this loop share
    //the same binding to v and p rather than having
    //distinct per iteration bindings.
  }
}
```

ES3

This pattern is quite common in code that manipulates the browser DOM—even experienced JavaScript programmers sometimes forget that a `var` declaration is not block-scoped.

The existing `var` declaration forms could not be changed to be block-scoped without breaking existing code. The $ES4_2$ effort had settled on using the keywords `let` and `const` for declarations that respected block scoping expectations. The keyword `let` was used to define mutable variable bindings and `const` for immutable constant binding. Their use was not restricted to blocks but could appear anywhere a `var` declaration could occur. The $ES4_2$ design group even had t-shirts made with the slogan "let is the new var." Harmony carried forward `let` and `const` declarations. But the $ES4_2$ work had left many issues relating to their semantics unanswered.

ES5 had contemplated adding `const` declarations and the ES5 specification included abstractions that could be used to specify block-level declaration binding semantics. But it was not obvious exactly what those semantics should be. The following code snippet illustrates some of the issues:

```
{ //outer block
  let x = "outer";
  { //inner block
    console.log(x);
    var refX1 = function() {return x};
    console.log(refX1());
    const x = "inner";
    console.log(x);
    var refX2 = function() {return x};
    console.log(refX2());
  }
}
```

ES2015

Should some or all of the references to `x` that occur within the inner block before the `const` declarations be compile time errors? Or, should they be runtime errors? If they are not errors should they resolve to the outer binding of `x`, or perhaps the inner `x` should have the value `undefined` until it is initialized? Does calling the function `refX1` before the `const` declaration resolve to the same binding of `x` and same value as when it is called after the declaration? All of the questions would still apply if the inner declaration of `x` was a `let` declaration. Waldemar Horwat [2008a] characterized four possible semantics for these references:

A1. Lexical dead zone. References textually prior to a definition in the same block are errors.
A2. Lexical window. References textually prior to a definition in the same block go to outer scope.
B1. Temporal dead zone. References temporally prior to a definition in the same block are errors.
B2. Temporal window. References temporally prior to a definition in the same block go to outer scope.

Horwat credits Lars Hansen for introducing the concept of "dead zones" to the discussion. The term "temporally prior" refers to the runtime evaluation order. A2 and B2 are undesirable because the same name at different points in the block can have different bindings and with B2 a name at a specific point in the block can have different bindings at different times. A1 is undesirable because it prevents these declaration forms from being used to define mutually recursive functions. A2 has the disadvantage that it requires run-time initialization checks on all references; however, many of them could be safely eliminated by a compiler using fairly simple analysis. It took nearly two years, but eventually the TC39 consensus was that the new lexical declaration forms should have the B1 temporal dead zone (TDZ) semantics. Those semantics are summarized by these rules:

- Within a scope, there is a single unique binding for any name.
- `let`, `const`, `class`, `import`, block level `function` declarations, and formal parameter bindings are dead at runtime until initialized.
- It is a runtime error to access or assign to an uninitialized binding.

In the specification, the first rule is expressed as an early-error rule, and the other two rules are expressed in the runtime semantic algorithms.

When Allen Wirfs-Brock began to integrate `let` and `const` into the specification he discovered many potential interactions, with legacy `var` and `function` declarations. This led to another round of TC39 discussions before reaching agreement on the following additional rules:

- Multiple `var` declarations for a name can exist at any level of block nesting. They all refer to the same binding whose definition is hoisted to the closest enclosing function or global top-level scope (ES1 legacy semantics).
- Multiple `var` and function/global top-level `function` declarations for the same name are allowed with one binding per name (ES3 legacy semantics).
- All other multiple declarations in a scope are early errors: `var`/`let`, `let`/`let`, `let`/`const`, `let`/`function`, `class`/`function`, `const`/`class`, etc.
- It is an early error for a block-level `var` declared name to hoist over any outer `let`, `const`, `class`, `import`, or block-level `function` declarations of same name.
- `var` declarations are auto-initialized to `undefined` when the binding is created so there is no TDZ limitation on accessing them.

Another set of issues concerned the handling of global declarations. Prior to ES2015, all global declarations created properties on the global object (§3.6) provided by the host environment. But object properties have no provisions for tagging a property as being uninitialized as is required to implement a temporal dead zone. One proposal was to treat global level occurrences of the new `const`, `let`, and `class` declarations as if they were `var` declarations. There was a precedent. Some

pre-ES2015 JavaScript engines had implemented `const` declarations in that manner. However, that would have created inconsistency between use of the new declarations at the global level and their use anywhere else. Instead, the TC39 consensus was that lexical declaration rules should apply as consistently as possible across all kinds of scopes. For the global scope, `var` and `function` declarations retain the legacy behavior of creating properties of the global object, however all other declaration forms create lexical bindings which are not backed by global object properties. The new rules disallowing conflicting `var`/`let` and similar conflicts apply except that global object properties which are not created using a `var` or `function` declaration do not result in multiple declaration conflicts. In those cases, a global `let`/`const`/`class` declaration shadows the like-named global object property. An implication of the rules is that globals defined using the new declarations cannot be multiply defined in separate scripts.

Simply adding block-scoped `let` and `const` declarations is not enough to fully eliminate the closure-in-loop hazard. There is also the problem of scoping the variable introduced by the `for` statement: `for (var p in x)`. ES2015 addresses this by allowing `let` and `const` to be used in the head of a for statement in place of `var`. A `let` or `const` used in this manner creates a binding in a scope contour that is recreated for each iteration of the loop body. The loop, `for (const p in x) {body}`, *desugars*[g] approximately as follows:

```
//approximate desugaring of: for (const p in x) {body}
{ let $next;
  for ($next in x) {
    const p = $next;
    {body}
  }
}
```

ES2015

Dealing with lexical bindings introduced by the C-style three-expression `for` statement is more complicated and was more controversial. JavaScript 1.0 had included the ability to use a `var` declaration as the first expression of such `for` statements, so a `let` or `const` declaration should also be usable there. But what is the extent of the bindings created by such declarations. Should there be a single binding whose lifetime is the duration of the entire `for` statement or should there be a separate binding for each iteration of the loop such as was done for the `for-in` statement? The answer is not obvious, because common coding patterns use the second and third expressions or code in the loop body to update the value of the declared loop variables for use on the next iteration of the loop. If each iteration gets a fresh binding of the loop variables it would be necessary to automatically use the final loop variable values from the previous iteration to initialize the loop variable bindings of the next iteration. Most C-like languages have used the single binding per `for` statement approach rather than the binding per iteration approach, and that is what the ES6 draft specification initially did. However, that approach still has the closure-in-loop hazard. For that reason, three-expression `for` statements with `let` declarations were eventually changed to use a binding per iteration with value propagation between iterations. A single binding per loop proved adequate for `const` declarations in the first expression, as such variables cannot have their values modified by the other expressions in the `for` header or the loop body.

Another significant issue concerned the semantics of functions declared within statement blocks. ES3 had intentionally excluded (§12) any syntax or semantic specification of function declarations within blocks. But implementations had ignored that guidance and allowed such declarations—unfortunately, each major-browser implementation gave them different semantics. However, there was enough semantic overlap that for some use cases [Terlson 2012] it was possible to declare such

functions and use them in a manner that would be interoperable among all major browsers. Some of those use cases would be illegal or change their meaning under the ES2015 lexical declaration rules. Implementing the new rules for those cases would "break the Web." This was not a problem for strict mode because ES5 had forbidden implementations from providing block-level function declarations within strict mode code. One approach for non-strict code would be to follow the example of ES3 and not specify anything about block-level functions—leaving it to each implementation to decide if and how to integrate block-level function declarations with the new lexical declaration forms. But that would not foster interoperability and would also be contrary to the 1JS goals [TC39 2013b]. Instead, TC39 [2013a] determined that there were only a few use cases where existing block-level functions were usefully interoperable and yet erroneous according to the new rules, for example:

```
function f(bool) {
  if (bool==true){
    function g() {/*do something*/}
  }
  if (bool==true) g(); //this worked in all major browsers
}
```

Non-standard but interoperable ES3 extension

The fix was to define some additional non-strict code rules [Wirfs-Brock 2015a, Annex B.3.3] that statically detect those specific interoperable use cases and make them legal and compatible with legacy Web pages. For the above example, the rules would treat the above code as if it had been code like this:

```
function f(bool) {
  var g;  //but early error if let declaration of g exists at top-level
  function $setg(v) {g = v}
  if (bool==true){
    function g() {/*do something*/}
    $setg(g); //set top-level g to value of local g
  }
  if (bool==true) g(); //references top-level g
}
```

ES2015+Annex B desugaring

21.3.3 Classes. At the July 2008 TC39 meeting that initiated the Harmony effort, considerable time was spent discussing whether and how classes should be included. Both ES4 efforts had put significant effort into developing a sophisticated class definition syntax and semantics, and both designs required new runtime mechanisms to support them. Those designs can be loosely characterized as "Java-inspired classes."

Mark Miller [2008d] argued that ES3 already had most of the runtime mechanisms necessary to implement class-like abstractions using lambda functions and lexical-capture techniques similar to those used in Scheme [Dickey 1992; Sussman and Steele Jr 1975] and adapted for JavaScript by Douglas Crockford [2008b, pages 52–55]. This style of lambda desugaring of class definitions is essentially the same as the module pattern (§13.2) suggesting that a class is simply a small lightweight module intended to be instantiated many times. Miller called this approach "classes as sugar."

Cormac Flanagan [2008] summarized the initial classes discussion, as follows:

> EcmaScript [*sic*] needs better support for providing high-integrity objects[92] with data abstraction and hiding, and for private fields and methods...
>
> ...We initially focus on a simple, minimalist design, with no support for inheritance or for type annotations, and with instance-private data. There is no separate namespace for class names, and class objects are a new kind of (first-class) values.

Flanagan's strawman used a simple syntax for class definitions, as follows:

```
class Point (initialX, initialY) {
  private x = initialX;
  private y = initialY;
  public getX() {return x};
  public getY() {return y};
}
```

Flanagan's Harmony Class Strawman

Cormac Flanagan's proposal lacked a full desugaring and included few semantic details. Mark Miller [2008c; 2009; 2010a] countered with a design with similar surface syntax. Miller's proposal included a complete desugaring which did not require a new kind of runtime object for class instances. In Miller's design there is no inheritance and all methods and instance variables default to private access. All methods and instance variables are represented as per-instance lexically captured declarations which are directly accessible from only within the body of the class definition. The properties of class instance objects provide external access to public methods, and `get` accessors are provided for public instance variables. Direct external external assignment to instance variables are not allowed. The `this` keyword is not used.

A recurring criticism of Mark Miller's classes-as-sugar proposals were that they created too many objects. Each object instantiation of a class with n methods implicitly creates n instance-specific closure objects in addition to the actual instance object. Miller's position was that the desugaring defined only the observable semantics and that implementations were free to develop techniques to avoid creation of the closure objects. However, committee skeptics doubted whether implementors would develop such optimizations. Another concern was the lack of support for inheritance or other behavioral composition mechanisms, so Miller also developed proposals [Miller 2010d, 2011a] that incorporated compositional Traits [Van Cutsem and Miller 2011c] into his class desugaring design.

Support for defining high-integrity objects was a top priority of those committee members who were most concerned about hostile Web ads and other Web mashups that might try to steal private information (§20.1.3). The entire committee shared this concern, but not necessarily the prioritization. Waldemar Horwat [2010] in his notes on the September 2010 TC39 meeting observed:

> Schism within group about goals: "high integrity" vs. "supporting the things that people are already writing with better syntax" vs. maybe possible to get both.

Allen Wirfs-Brock believed that making object creation less imperative might support the second goal. In classic JavaScript, the closest analog to a Class is a constructor function (§3.3) which imperatively defines a new object's properties. Object literals provided a more declarative way to define an object's properties but lacked the affordances needed to easily match the conventions[93] followed for ECMAScript's built-in Classes. Perhaps object literals could be extended to better

[92] A "high-integrity object" supports impenetrable information hiding. Its structure, inheritance relationships, encapsulated state, and methods cannot be directly modified or extended other than by the code that defines the object.

[93] For example, the non-enumerability of methods and the use of read-only properties.

```
function tripleFactory(a,b,c) {
  return { // This object literal creates the triple objects
    <proto: Array.prototype, // proto meta-property sets inheritance
        prototype
    sealed>,                 // sealed meta-property applies Object.seal()
    0: a,
    1: b,
    2: c,
    var length const:3, // var sets [[enumerable]]: false,
    // const: sets [[writable]]: false
    method toString(){  // methods are data properties with function values
      // and [[enumerable]]: false
      return "triple("+this[0]+","+this[1]+","+this[2]+")"},
    method sum(){return this[0]+this[1]+this[2]}
  }
}
```

Fig. 45. A factory function based on Wirfs-Brock [2011c; 2011d] Harmony Extended Object Literal Proposals

support what people were already writing without having to introduce "classes" as new kind of language entity.

In a group of related proposals, Wirfs-Brock [2011c; 2011d] showed how object literals might be extended to be more declarative and eliminate the need to use the ES5 object reflection APIs for routine object-definition use cases. For example, Figure 45 shows how a *factory function*[8] that uses extended object literal features would define classes with an explicit prototype, methods, and private properties.

Allen Wirfs-Brock's proposals also showed how the extended object literal syntax could be used as the body of a class definition. In a March 2011 TC39 presentation, Wirfs-Brock [2011a] proposed that class definitions should generate the basic triad of constructor function, prototype object, and instance objects used for the built-in library Classes in clause 15[94] of the ECMAScript specifications of all previous editions of ECMA-262. Rather than desugaring class definitions into lambda expressions (classes as sugar) or a new kind of runtime entity (Java-inspired classes), they should desugar into the familiar constructor functions and prototype-inheriting objects already used by JavaScript programmers and framework authors. At the meeting there were significant differences of opinion about many details of the extended object literal syntax, but a loose consensus was established that the core class definition semantics should be the clause 15-constructor/prototype/instance triad.

In early May 2011, the TC39 ES.next feature-freeze meeting was rapidly approaching and there were still several competing strawman proposals relating to classes. It seemed doubtful that there would be sufficient consensus for any of the proposals to make the cut. On May 10, 2011, Allen Wirfs-Brock met with Mark Miller, Peter Hallam, and Bob Nystrom. Hallam and Nystrom were members of the team that was prototyping JavaScript class support using Google's Traceur transpiler [Traceur Project 2011b]. Their prototype incorporated ideas from both Wirfs-Brock's and Miller's proposals. The goal of the meeting was to get sufficient agreement to be able to present a unified proposal. Bob Nystrom [2011] in his meeting report lists many points of agreement including:

> …The trinity of constructor functions, prototypes, and instances are more than adequate for solving the problems that classes solve in other languages. The

[94]Prior to ES6, clause 15 was the section of the specification that defined the built-in objects and Classes.

> intent of a Harmony class syntax is not to change those semantics. Instead, it's to provide a terse and declarative surface for those semantics so that programmer intent is shown instead of the underlying imperative machinery.
>
> …Objects are declarative and informational. Functions are imperative and behavioral. The question with classes is, "do we build on one of those abstractions and if so, which one?"…
>
> Our consensus proposal addresses this religious disagreement by using both: an object literal-like form for the class body itself, and a function for the constructor.

After the meeting, Mark Miller [2011b] created a new strawman that was accepted at the feature-freeze meeting [TC39 2011b] even though there remained many details of the proposal that lacked consensus. The example class definition in Figure 46 is based upon one given in Miller's feature-freeze class proposal.

A month later, Dave Herman [2011c] in an es-discuss post titled "minimal classes" expressed concerns that the complexity of the class proposal and its many points of disagreement created a schedule risk for ES.next. He suggested an alternative minimized design which included only class declarations with prototype inheritance, constructors, declarative methods, and calling inherited methods using the `super` keyword. Excluded would be declarative properties, constructor properties, private data, and anything else that was controversial. Herman's suggestion was discussed at the July 2011 meeting [TC39 2011a], but the committee decided to focus on resolving the open issues of Mark Miller's then-current proposal. Brendan Eich [2012a] later wrote:

> Minimal classes had a good subset of TC39 supporting last summer in Redmond, but we got hung up future-proofing use-before-initialization for const and guards…

Continuing online discussions about alternative class designs [Ashkenas 2011; Eich 2011a; Herman 2011a] motivated Dave Herman [2011d] to write a new "minimal classes" strawman.

```
class Monster extends Character {
  constructor(name, health) { //the constructor function
    super();             //call superclass constructor
    public name = name;  //a public instance property
    private health = health; //a private instance variable
  }
  attack(target) {  //a prototype method
    log('The monster attacks ' + target);
  }
  get isAlive() {  //a prototype get accessor
    return private(this).health > 0;
  }
  set health(value) { //a prototype set accessor
    if (value < 0) {
      throw new Error('Health must be non-negative.')
    }
    private(this).health = value
  }
  public numAttacks = 0; //a prototype data property
  public const attackMessage = 'The monster hits you!'; //read-only
}
```

Fig. 46. A class declaration based on Mark Miller's [2011b] Unified Harmony Class Proposal

This proposal formalized Herman's earlier post but added "static" constructor data and method properties. There was little discussion of Herman's minimal proposal at the next two TC39 meetings and little progress toward resolving disagreements about the plan of record. Brendan Eich [2012c] described the problem as follows:

> ...the general tendency observed by Waldemar [Horwat] is real: too minimal and there's no point. Too maximal and we can't agree. We need "Goldilocks classes"—just the right temperature/amount.

By early March of 2012, es-discuss community members were expressing growing frustration with the apparent inability of TC39 to finalize a design for ES.next classes. Russell Leggett [2012] in a post titled "Finding a 'safety syntax'[95] for classes" asked the question:

> Is it possible that we can come up with a class syntax that we can all agree is better than nothing, and importantly, leaves possibilities open for future enhancement? As a "safety syntax" this doesn't mean we stop trying to find a better syntax, it just means that if we don't find it then we still have something—something that we can make better in ES7.

Leggett's post received 119 predominantly positive responses over three days. It listed a set of "absolute minimal requirements" that was essentially the same as Dave Herman's list from the previous summer. Leggatt's contribution was the safety school metaphor. Allen Wirfs-Brock immediately expressed his support and created a new "maximally minimal" version [Wirfs-Brock 2012d] of Herman's Minimal Classes proposal reframed using that metaphor. The most significant technical change was to remove constructor properties from the proposal.[96] It was too late to officially place the "max-min" proposal on the agenda for March 2012 TC39 meeting, but Allen Wirfs-Brock and Alex Russell led an informal discussion at the end of the meeting [TC39 2012a]. The reception was generally positive, but a few members expressed concerns that the proposal might be too minimal to bother with or that it might be hostile to future extensions that they contemplated. There was no attempt to reach a consensus on the proposal, but Wirfs-Brock and Russell expressed the opinion that anything more elaborate was unlikely to make it into ES.next.

The max-min proposal was officially on the agenda for the May 2012 meeting and a similar discussion with similar results took place [TC39 2012b]. Those present were inching toward consensus on the proposal but some key individuals were absent. Because of schedule pressure, there was agreement that it was acceptable to work on prototypes and preliminary specification drafts. By the time of the July meeting [TC39 2012c], Allen Wirfs-Brock had written specification text for max-min classes and prepared a presentation deck [Wirfs-Brock 2012b] that enumerated every design decision he encountered. He walked the committee through a review of each decision and recorded either acceptance or consensus on an alternative. This approach sidestepped the question of consensus on the entire proposal but got the committee engaged in consensus formation at the detailed design level. The next draft [Wirfs-Brock et al. 2012b,c] of the ES.next specification incorporated the complete max-min class design including the decisions made at the July meeting. Nobody objected.

However, in the summer of 2014, as browser JavaScript engine developers started working on implementations of ES6 classes, a significant objection did appear. A long standing goal of the ES6 effort was to provide a means of "subclassing" built-in Classes such as `Array` [Kangax 2010] and the Web platform DOM Classes. Allen Wirfs-Brock [2012c; 2012e] wrote a Harmony Strawman that describes why traditional JavaScript approaches to subclassing built-in constructors were

[95] Russell Leggett was making an analogy with a "safety-school," which a perspective college student applies to as a backup in case they are not admitted to any of their preferred schools.

[96] Constructor methods were eventually added back to the design using the `static` keyword.

problematic. Built-in constructors are typically defined using an implementation language such as C++. They allocate and initialize private object representations whose unique structure is known to the associated built-in methods which are also defined using the implementation language. This works when a built-in constructor is directly invoked using the `new` operator, but when "subclassing" such constructors using JavaScript's ad hoc prototype inheritance scheme, the `new` operator is applied to the subclass constructor (typically coded in JavaScript) which allocates an ordinary object rather than the private object representation expected by the inherited built-in methods. Wirfs-Brock [2013] attempted to avoid this problem when specifying the max-min class semantics. The semantics of `new` was split into separable allocation and initialization phases. Object allocation was performed by `new` first invoking a specially named `@@create` method that would typically be provided by a built-in superclass and not overridden by subclasses. Object initialization occurred after allocation and was orchestrated by the subclass constructor. It would typically make a `super` call to its superclass constructor to perform any necessary superclass-specific initialization and then perform any subclass-specific initialization. When properly coded, this enables a built-in superclass to allocate its unique private object structure before passing the object to the subclass constructor which can use its initialization code to add subclass properties to the superclass-provided object.

The problem that was identified in 2014 was that the objects created by `@@create` methods are uninitialized, and a buggy or malicious class constructor might invoke a built-in superclass method (likely implemented in C++) on an uninitialized object—probably with catastrophic results. Wirfs-Brock had assumed that all such objects would internally keep track of their initialization state and that the corresponding built-in methods would be required to check if they were being applied to an uninitialized object. Mozilla's Boris Zbarsky [2014] pointed out that browsers have thousands of such methods, and the two phase approach would require updating for each method the DOM specifications and implementations for every browser. This motivated development of a single phase allocation/initialization approach [Wirfs-Brock et al. 2014c,d] and another proposal [Herman and Katz 2014] which retained the two phases but passed the constructor arguments to both the `@@create` method and the constructor. These and other alternatives were hotly debated throughout the remainder of 2014, and for awhile it appeared that a lack of consensus might either delay the planned June 2015 publication of ES6 or force complete removal of classes from that edition. However, in January 2015 a TC39 consensus formed around a variation of the single phase approach [TC39 2015a; Wirfs-Brock 2015b]. This experience reïnforced TC39's resolve to require more and earlier implementor feedback on post-ES6 new features.

21.3.4 Modules. A complex aspect of the ES4 designs had been the package and namespace constructs for structuring large programs and libraries. By the time $ES4_2$ was abandoned, significant problems had been identified [Dyer 2008b; Stachowiak 2008b] with those mechanisms, and it was clear that they would not be suitable for Harmony. At that time, ad hoc modularity solutions based on the module pattern (§13.2) were being adopted by influential JavaScript developers [Miraglia 2007; Yahoo! Developer Network 2008]. In January 2009, Kris Kowal and Ihab Awad presented a module-pattern–inspired design [Awad and Kowal 2009; Kowal and Awad 2009a] to TC39 [2009c]. Their design eventually evolved to become the CommonJS module system used by Node.js.

Kris Kowal and and Ihab Awad, in their original proposal and a subsequent revision [Kowal 2009b; Kowal and Awad 2009b], included syntactic-sugaring alternatives that might overlay their module design without changing the proposal's dynamic semantics. Awad [2010a; 2010c] then developed a different proposal that drew from both the CommonJS work and the Emaker modules of the E Language [Miller et al. 2019] that were being used by the Secure ECMAScript Caja Project [2012]. Within TC39, these proposals were called "first-class module systems" because they manifest modules as dynamically constructed first-class runtime entities that provide a new computational

abstraction mechanism. For example, in Awad's proposal, multiple instances of a module may simultaneously exist, each initialized with different parameter values.

Brendan Eich [2009c] described an alternative approach:

> The alternative for Harmony is a syntactic special form, an import directive for example, that can be analyzed when the program is parsed (not executed), so the implementation can preload all dependencies before execution to avoid blocking on an import (or a later data dependency), or else an awkward non-blocking import to preserve JS's run-to-completion execution model.

This alternative approach was called a "static" or "second-class module" system. A second-class module system provides mechanisms for structuring application code rather than mechanisms for defining new computational abstractions. Sam Tobin-Hochstadt [2010] explained:

> . . . in a language with state, you want to be able to divide your program up into modules without changing its behavior. If you have a stateful piece of code, and you move it into its own module, that shouldn't change things any more than any other refactoring. If you need to repeatedly create fresh state, ES provides nice mechanisms for that as well. Similarly, if you have one module that imports A, and you divide it into two, both of which now import A, that refactoring shouldn't change how your program works.

Dave Herman and Sam Tobin-Hochstadt developed a "Simple Modules" design [Herman 2010b,c,f; Herman and Tobin-Hochstadt 2011; Tobin-Hochstadt and Herman 2010] for second-class Harmony modules. The basic idea was that modules were units of code that could share lexical bindings. Syntax was used to delimit the units of code and to identify which bindings would be shared. TC39 extensively debated the merits of the two approaches until Awad [2010b] recommended that TC39 focus its efforts on the Herman/Tobin-Hochstadt proposal.

Their design had `module` declarations that assigned a lexical identifier to the module and either included the module code or identified an external resource containing the code. An `export` keyword prefixed declarations whose binding were to be exposed outside of the module, for example:

```
module m1 {          // an internal module
  export var x = 0, y=0;
  export function f() {/* ... */};
}
module m2 {          // another internal module in same source file
  export const pi = 3.1415926;
}
module mx = load "http://example.com/js/x.js";
            // String literal identifies an external module
// ... code that imports and uses bindings from m1, m2, and mx.
```

Original Harmony Simple Modules Proposal

Module declarations could also be nested. An external module such as `x.js` could consist of only a module body without the surrounding module-declaration syntax. An `import` declaration is used to make a binding exported by a module lexically accessible to an importing module. Code that uses the above example modules might have `imports` like the following:

```
import m1.{x, f};  //import two exported bindings from m1
import m2.{pi: PI};//import a binding and rename it for local access
import mx.*;       //import all of the bindings exported by mx
import mx as X;    //Locally bind X to an object whose properties bind
                   //to the exports of mx
```

Original Harmony Simple Modules Proposal

Module declarations, string literal external module specifiers, and declarative export/import definitions enable static determination of a closed set of interdependent modules whose shared lexical bindings can be linked prior to the execution of any code. Circular dependencies are allowed. When execution starts, modules are initialized in a specified deterministic order and temporal dead zones ensure run-time errors if any circular dependencies are impossible to initialize.

The syntax evolved [Herman et al. 2013], but the basic idea of statically linkable modules with shared lexical bindings remained. A major change was the elimination of explicit module declaration syntax, module identifiers, and internal/nested modules. Harmony modules are defined one per source file and are identified using literal string resource specifiers. The elimination of module identifiers required changing the `import` syntax, and wildcard imports were eliminated as being too error prone. Wildcard imports were replaced with an alternative form that exposes an open-ended set of imports as properties of a single namespace object rather than as individual lexical bindings. The above `import` examples expressed using the final syntax look like this:

```
import {x, f} from "m1.js";     //import two exported bindings from m1
import {pi as PI} from "m2.js";//import a binding. Rename it for local access
import * as X from "mx.js";     //Locally bind X to a namespace object whose
                                //properties (*) map to the exports of mx.js
//additional import forms
import from "my.js";    //import my.js only for initialization effects
import z from "mz.js"; //import the single default binding exported by mz.js
```

ES2015

The elimination of `module` declarations and the addition of the default binding `import` form were a late change to the design. Node.js adoption had been unexpectedly rapid, and it had widely exposed CommonJS modules to the JavaScript developer community. TC39 was getting negative community feedback [Denicola 2014] and had concerns that de facto standardization of CommonJS modules might overshadow the Harmony design. The `export default` form was added to accommodate developers who were accustomed to the single export pattern[97] used in many CommonJS modules. TC39 module champions also began to evangelize [Katz 2014] Harmony modules to Node.js developers.

The initial Simple Modules proposal had included the concept of a module loader [Herman 2010e] which provided the semantics of incorporating modules into a running JavaScript program. The intent was that the ECMAScript specification would define the language level syntax and semantics of modules, the runtime semantics of module loading, and a module loader API which would provide JavaScript programmers with mechanisms to control and extend the loader semantics. The loading process was ultimately envisioned [Herman 2013b] to be a pipeline consisting of five stages: normalize, resolve, fetch, translate, and link. The loader begins by normalizing a module identifier. It then progresses through the retrieval and preprocessing of module source code, determining module interdependencies, linking imports to exports, and finally initializing the interdependent modules. The intent was that the module loader would be extremely flexible and fully support the asynchronous I/O model of Web browsers. At JSConf 2011, Dave Herman demonstrated [Leung 2011] a proof-of-concept module loader which extended the translation stage to load CoffeeScript and Scheme code as modules of a JavaScript-based Web page.

In order to fully understand the module loading process and how to specify it, Dave Herman worked with Jason Orendorff at Mozilla to prototype a module loader reference implementation [Orendorff and Herman 2014] using JavaScript code. In December 2013, Herman [2013a] completed an initial rewrite of Orendorff's JavaScript code into specification pseudocode, and in

[97] CommonJS modules typically export a single object that is used as a namespace.

January 2014 Allen Wirfs-Brock [2014a] had a preliminary integration of the pseudocode into the ES6 draft. Wirfs-Brock found that the asynchronous nature of the module loader added significant new complexity and potential nondeterminism to the ECMAScript specification. This was made worse by the loader API which allowed user programs to inject arbitrary JavaScript code into the module loading process. By the middle of 2014, the additional complexity of asynchronous module loading and a stream of difficult-to-resolve design issues with the API appeared to put the 2015 target release of ES6 in jeopardy.

Early in the development of the simple module proposal, Allen Wirfs-Brock [2010] had observed that the semantics of module scoping and linking could be separated from the loader pipeline. Previous editions of ECMA-262 had defined the syntax and semantics of JavaScript source code but did not address how it was accessed. That was left as a responsibility of the environments that hosted a JavaScript engine. At the September 2014 TC39 meeting [TC39 2014b], Wirfs-Brock argued that a similar approach could be used for modules. ECMA-262 did not need to include the specification of a module loading pipeline. If ECMA-262 assumed that the source code for modules was already available, it would be sufficient to specify the syntax and semantics of individual modules and the semantics of linking imported bindings to exported bindings. A host environment, such as a browser, could provide an asynchronous loading pipeline but its definition would be decoupled from the language specification. Removing the loader pipeline also implied the removal of the loader API. TC39 accepted this argument, and Wirfs-Brock was able to incorporate a nearly complete language-level specification of modules into the October 2014 specification draft [Wirfs-Brock et al. 2014b]. The separation of module semantics from the loader pipeline enabled the WHATWG to focus on specifying how ECMAScript modules would integrate with the Web platform [Denicola 2016].

21.3.5 Arrow Functions. ES2015 introduces a concise form of function definition expressions commonly called "arrow functions." An arrow function is written as a formal parameter list followed by the token => followed by a function body, for example:

```
(a, b) => {return a+b}
```

If there is only a single parameter, the parentheses may be omitted and if the body is a single `return` statement, the braces and the `return` keyword may be omitted, for example:

```
x => x /* an identity function */
```

Unlike other function definition forms, arrow functions do not rebind `this` and other implicit function-scoped bindings. This makes arrow functions convenient in situations where an inner function needs full access to the implicit bindings of its outer function.

The primary motivation for arrow functions was the frequent need to code verbose function expressions as call-back arguments to platform and library API functions. In JavaScript 1.8, Mozilla [2008b] had implemented[98] "expression closures" which retained use of the `function` keyword and permitted only a brace-free single expression body. TC39 discussed similar shorter formulations, which replaced `function` with symbols such as λ, *f*, \, or # [Eich 2010b; TC39 Harmony 2010c], but could not reach consensus on any of them.

There was also interest within TC39 [Herman 2008] in providing "lambda functions" with streamlined semantics such as support for *proper tail calls*[g] and Tennent's [1981] correspondence principle.[99] The proponents argued that such functions would be useful for implementing both language- and library-defined control abstractions. In an es-discuss post early in the Harmony

[98] Based upon an $ES4_2$ proposal [TC39 ES4 2006c]

[99] Within a function, wrapping a code sequence with another function that is immediately called should produce the same effect as directly executing the original code sequence.

process, Brendan Eich [2008a] floated a suggestion originally made by Allen Wirfs-Brock for a concise lambda-function syntax inspired by Smalltalk block syntax. For example, `{|a, b| a+b}` would be equivalent to Herman's `lambda(a,b){a + b}`. Eich's post initiated a massive but inconclusive electronic discussion covering all aspects of a possible concise function feature. Some key takeaways were that many of the syntax ideas presented parsing or usability issues and that JavaScript's nonlocal control transfer statements—`return`, `break`, and `continue`—significantly complicated mechanisms for writing control abstractions. Most TC39 members and es-discuss subscribers seemed interested primarily in concise function syntax rather than in Tennent's correspondence.

No significant progress was made made for 30 months until Brendan Eich [2011f; 2011g] wrote two alternative strawman proposals. One was for "arrow functions" modeled on a similar feature in CoffeeScript. This proposal had both -> and => functions with various syntactic and semantics differences and options. The other proposal was for "block lambdas" modeled on Smalltalk and Ruby blocks and supported Tennent's correspondence. Over the following nine months both proposals and alternatives were extensively discussed on es-discuss and at TC39 meetings. There were concerns about whether existing JavaScript implementations could be easily updated to parse arrow functions. The problem was that the arrow symbol occurs in the middle of the construct and is preceded by a parameter list that could be ambiguously parsed as a parenthesized expression. For the Block Lambda proposal there were concerns [Wirfs-Brock 2012a] that it did not adequately support creating user-defined control structures that fully integrated with the built-in syntactic control structures. Brendan Eich generally preferred the Block Lambda proposal but as the March 2012 TC39 meeting approached, he concluded that arrow functions were more likely to be accepted by the committee. At the meeting [TC39 2012a] he walked the committee through a set of consensus decisions on the essential characteristics of the final arrow function design [Eich 2012b].

21.3.6 Other Features. In addition to those already discussed, significant new language features include:

- object literal enhancements including computed property names and concise method syntax
- object and array destructuring in declarations initializers and assignment operators
- formal parameter enhancements including rest parameter, optional parameter default values, and argument destructuring
- iterators and generators—inspired by Python but with significant differences
- `for-of` statement and pervasive use of iterator protocol in new and retrofitted contexts
- full Unicode supported in strings and regular expressions
- template literals supporting embedded domain specific languages
- `Symbol` values for use as property keys
- binary and octal numeric literals
- Proper Tail Calls[100]

Library enhancements include:

- new `Array` methods
- `of` and `from` constructor method conventions for creating Arrays and other collection objects.
- Typed array Classes including `DataView` and `ArrayBuffer` for manipulating binary data; all based on a Khronos Group [2011] specification previously implemented as browser host objects but with better integration with the rest of the language; Typed Arrays now support most `Array` methods
- `Map` and `Set` keyed collections and `WeakMap` and `WeakSet`

[100]PTC has proven to be a contentious feature. It has been successfully implemented by at least one major browser engine but others have refused to support it.

- additional `Math` and `Number` functions
- `Object.assign` function for copying object properties
- `Promise` class for deferred access to asynchronously computed values
- `Reflect` functions reifying the internal metaobject protocol

21.3.7 Deferred and Abandoned Features. Over the course of ES6 development, many strawman feature proposals were considered by TC39 but ultimately were not included as a feature of ES2015. Many of these were rejected soon after their initial presentation, but others were the subject of significant development work, and some even advanced to the status of accepted Harmony proposal before ultimately being cut from the release. Of the cut features, some were abandoned and others were deferred for additional work and possible consideration for inclusion in future editions. Major features and development efforts cut shortly before the completion of ES2015 include the following:

Comprehensions [Herman 2010a,d, 2014a; TC39 2014a] Based upon a similar features in Python and JavaScript 1.7/1.8, comprehensions would have provided a more concise, declarative way to create an initialized array or to define a generator function.

Module loader API [Herman 2013b] The module loader API would have enabled a JavaScript programmer to dynamically intercede in the processing performed by the Module Loader. A program might use the API to do things such as inserting a transpiler into the loading process or support the dynamic definition of modules. This API was deferred along with the Module Loader.

Realms API The Realm API [Herman 2014b] would have enabled JavaScript programmers to create, populate, and execute code in new Realms. It was closely related to the Module loader API and deferred for additional design work.

Pattern matching [Herman 2011e; Rossberg 2013] A generalization of destructuring that would have included Haskell-inspired refutable matching.

Object.observe [Arvidsson 2015; Klein 2015; Weinstein 2012] A complex data binding mechanism that could generate events when properties of monitored objects were modified.

Parallel JavaScript Also known as River Trail [Hudson 2012, 2014]. A joint project of Intel and Mozilla intended to enable JavaScript programmers to explicitly exploit the SIMD capabilities of processors.

Value Objects [Eich 2013] Generalized support, including operator overloading, for defining primitive data types similar to Number and String. Potentially could allow libraries to implement decimal numbers, large integers, etc.

Guards [Miller 2010c] Type-like annotations on declarations that would be dynamically validated.

21.4 Harmony Transpilers

Transpilers played an important role in the development, testing, and community socialization of Harmony features. They enabled production use of new features prior to completion of the standard or its full support in browsers. Transpilers were essential to the rapid adoption of ES2015 by the JavaScript developer community. Important transpilers supporting Harmony included the following:

Narcissus [Eich et al. 2012] is a JavaScript-hosted JavaScript engine that was used by Mozilla Research for ES6 language experimentation.

Traceur [Hallam and Russell 2011; Traceur Project 2011a] is a transpiler developed by Google and used for experimenting with early ES6 features. Traceur provided a high-fidelity implementation of ES6 semantics, but the resulting runtime overhead made it unattractive for production use.

Allen Wirfs-Brock (Project Editor)	Microsoft, Mozilla
Brendan Eich	Mozilla, invited expert
Mark S. Miller	Google
Waldemar Horwat	Google
Dave Herman	Northeastern Univ, Mozilla
Douglas Crockford	Yahoo!, PayPal
Erik Arvidsson	Google

Fig. 47. TC39 technical contributors who were active throughout the ES2015 development effort. Each person attended at least 30 of the 41 TC39 meetings during that period. Arvidsson first attended in May 2009. Crockford last attended in April 2014. The rest participated from the beginning to the end of the project.

Babel [2015] originally named 6to5, was developed by Sebastian McKenzie, then a 17-year-old developer living in rural Australia: "On September 28th 2014 I pushed my first commit to GitHub for a JavaScript library I was working on while studying for my high school exams." [McKenzie 2016] Babel minimized runtime overhead by sacrificing complete semantic conformance to the draft specification. It offered early access to ES2015 features and other experimental JavaScript features enabling most ES2015-level JavaScript code to run on older browsers or platforms that support only ES5. However, some developers using Babel became dependent on experimental features, incorrect semantics, or obsolete variants of what later become standard ECMAScript features. This made the transition to native implementations harder and in a few cases has created a legacy that limits TC39's design flexibility.

TypeScript [Microsoft 2019] is a free Microsoft language product that originally targeted ES5 with ES6+ features and later added ES2015 as a compilation target. TypeScript's most important feature is an optional statically analyzable type system and type annotations which compile into idiomatic dynamically typed JavaScript code. In 2020, TypeScript is the de facto standard for writing type-annotated JavaScript [Greif and Benitte 2019].

The production use of transpilers, especially Babel and Typescript, was part of a large cultural transformation within many JavaScript development teams. In those teams, JavaScript is treated similarly to a conventional, ahead-of-time compiled language with development and deployment build toolchains rather than as a dynamic execution environment that loads and directly executes a programmer's original source code.

21.5 Finishing ECMAScript 2015

At its March 2015 meeting, TC39 [2015b] approved the then-current candidate specification [Wirfs-Brock et al. 2015b,c] and referred it to the Ecma General Assembly for final approval. The Ecma GA voted to accept it [Ecma International 2015a] at its June 2015 meeting and immediately published ECMA-262, 6th Edition, titled *ECMAScript 2015 Language Specification*[101] [Wirfs-Brock 2015a].

It took nearly seven years to develop and release ECMAScript 2015 and hundreds of people contributed to its development. There were 41 TC39 meetings starting with the July 2008 meeting (where the Harmony effort began) through the March 2015 meeting (where the candidate specification was approved). These meetings were attended in person or via teleconference by 145 people with varying levels of participation. ES2015 development overlapped with development of ES5/ES5.1, *ECMA-402 The ECMAScript Internationalization API*, *ECMA-404 JSON Interchange Format*, and the Test262 validation test suite. The primary interest of some attendees was one or

[101] TC39 added the year to the title because it planned to follow Edition 6 with incremental yearly updates. The hope was that the JavaScript community would begin to refer to specification revisions by year rather than by edition number.

Sam Tobin-Hochstadt (24)	Andreas Rossberg (13)	Rafael Weinstein (10)	Chris Pine (7)
Alex Russell (21)	Oliver Hunt (12)	Jeff Dyer (8)	Mike Samuel (6)
Luke Hoban (20)	Norbert Lindenberg (12)	David Fugate (8)	Ihab Awad (5)
Cormac Flanagan (18)	Sam Ruby (12)	Domenic Denicola (7)	Reid Burke (5)
Yehuda Katz (17)	Brian Terlson (12)	Rick Hudson (7)	Andreas Gal (5)
Rick Waldron (17)	Sebastian Markbage (11)	Jafar Husain (7)	Peter Jensen (5)
Eric Ferraiuolo (15)	Jeff Morrison (11)	Dimitry Lomov (7)	Pratap Lakshman(5)
Tom Van Cutsem (14)	Rob Sayre (10)	Ben Newman (7)	Nicholas Malsakic (5)
Nebojsa Ćirić (13)	Matt Sweeney (10)	Caridy Patino (7)	

Fig. 48. Technical contributors who frequently participated in TC39 meetings during the development of ES2015. The numbers reflect how many meetings they attended.

more of those efforts. Of the 145 meeting attendees, 62 individuals attended only a single meeting, typically as observers.

TC39 chair John Neumann and Ecma Secretary-General István Sebestyén provided administrative support and ensured that meetings ran smoothly. The project editor, Allen Wirfs-Brock, released 38 drafts of the specification [TC39 Harmony 2015] over the course of the project. Seven people (Figure 47) were technical contributors over essentially the entire project. An additional 35 participants (Figure 48) attended between 5 and 24 meetings with most making significant technical contributions to the project. Over the course of ES2015 development hundreds of members of the JavaScript developer community posted over 36,000 messages to the es-discuss mailing list [TC39 et al. 2006]. Over 4,000 tickets relating to the ES2015 specification drafts were opened in the TC39 bug tracking system [TC39 et al. 2016].

Interest and participation in TC39 grew dramatically during the development of ES6 and continued after its completion. TC39's first Harmony meeting in July 2008 was attended by only 13 individuals representing 8 organizations. The July 2015 meeting, held a month after publication of ES2015, had 34 individual participants (some remote) representing 15 organizations. The July 2019 TC39 meeting had 76 participants (46 local and 30 remote) representing 24 organizations.

21.5.1 Preparing for the Post-ES6 Future. In 2013 and 2014, as the conclusion of ES6 development approached, TC39 started to consider how development of future editions should proceed. One concern about the ES6 process was that the design of some features were completed several years before they could appear in a published ECMAScript standard. This was in conflict with the concept of "evergreen browsers" that was being adopted by most major browser developers. Evergreen browsers are updated every few weeks, making bug fixes and new features available as soon as possible. Most TC39 members felt that there was a need for a much faster update cycle for the ECMAScript standards which would better match the rapid evolution of browsers.

A yearly publication cycle was proposed. This would allow individual new features to become quickly available in a standard. Yearly releases would also allow specification bugs to be quickly corrected and eliminate the need to maintain a long errata spanning many years. A yearly publication update cycle was extremely fast by the norms of standards organizations, but was a schedule that Ecma agreed to accommodate.

Yearly updates would require TC39 to be more disciplined in how it developed new language features. Some design efforts would still require multiple years to complete, so a process was needed that could accommodate feature development projects that spanned multiple yearly release cycles and could coordinate overlapping development cycles for different features. There were also concerns that ES6 had depended too much on a single editor to do most of the specification writing.

To succeed with yearly releases, champions would need to do most of the specification writing for their features.

Rafael Weinstein and Dimitry Lomov presented a proposal [TC39 2013c; Weinstein and Lomov 2013] for a development process where new feature proposals progressed through five maturity stages. Weinstein then worked with Allen Wirfs-Brock to further define and document the process [Weinstein and Wirfs-Brock 2013]. Appendix P is the description of the new process and the development stages. Starting in 2014, TC39 followed this process for all of its post-ES6 efforts. As of the June 2020 publication of this paper, TC39 has successfully published an updated edition of the ECMAScript specification each June.

22 CONCLUSION

JavaScript was a language created with low expectations. It was originally intended to be a sidekick to Java within browsers, suitable for beginner Web page developers and part-time programmers. Yet, in short order, it surpassed Java as the primary language for interactive Web pages. Even though JavaScript's first 20 years is littered with failed attempts to enhance, improve, redesign, or replace it, by the end of that period JavaScript was the world's most widely used programming language—and not only for Web pages. In addition to server applications built using Node.js and other hosts, JavaScript is being used to build desktop applications, mobile device applications, fitness trackers, robots, and numerous embedded systems. It is even part of the James Webb Space Telescope which uses Nombas' ES1 level embedded JavaScript as part of its on-board control software [Dashevsky and Balzano 2008].

Was the rise of JavaScript inevitable? The interoperability requirements of the Web and Browser Game Theory may favor a single dominant Web page programming language, but there was no singular reason that the language had to be JavaScript. Other languages could have filled that role. Indeed, there are many places in the history of JavaScript where the outcome could have been different:

What if Marc Andreessen had not championed development of a browser scripting language?
What if Sun's Bill Joy had not supported the initial Mocha effort as complementary to Java?
What if the task of developing Mocha had be given to someone other than Brendan Eich?
What if Eich had been a more experienced language designer or implementor, and concluded that the 10-day demo was an impossible task?
What if Eich had failed in creating the 10-day Mocha demo, either because he was a less capable programmer or too ambitious in his language design?
What if JavaScript's original design had not included first class functions?
What if Sun or Netscape had put effort into better integrating Java with HTML, instead of hosting Java as an isolated environment?
What if Microsoft had not implemented JScript, but instead more strongly promoted its Visual Basic alternative?
What if Microsoft had continued investing in browser language technologies after achieving over 90% browser market share?
What if Macromedia/Adobe had pushed to make ActionScript 2 or 3 an official browser standard, rather than participating in the $ES4_2$ redesign?
What if opposition to $ES4_2$ had not emerged within TC39?

What if, what if, what if... but none of these things actually happened. Instead, sometimes facing stiff criticism and even ridicule, a generation of browser implementors, engine developers, framework designers, standards contributors, tool builders, and Web application programmers found pragmatic ways to use and enhance JavaScript, usually without breaking the Web.

Brendan Eich characterized JavaScript in a 2011 conference talk titled "JSLOL" [Eich 2011e]:

- First they said JS couldn't be useful for building "rich Internet applications"
- Then they said it couldn't be fast
- Then they said it couldn't be fixed
- Then they said it couldn't do multicore/GPU
- Wrong every time!

My advice: **Always bet on JS**

ACKNOWLEDGMENTS

Members of the HOPL-IV program committee assisted the authors (Figure 49) with shepherding, LaTeX hacking, detailed reviews, and valuable feedback on drafts of this paper.

The following colleagues, who participated in the development of JavaScript and ECMAScript, provided information on events and technologies discussed in this paper: Douglas Crockford, Jeff Dyer, Richard Gabriel, Bill Gibbons, Gary Grossman, Lars T. Hansen, Dave Herman, Graydon Hoare, Yehuda Katz, Shon Katzenberger, Peter Kukol, Pratrap Lakshman, Mark S. Miller, István Sebestyén, Mike Shaver, Brian Terlson, Tom Van Cutsem, Herman Venter, Rick Waldron, and Robert Welland.

Beta readers who provided editorial feedback on some or all of the manuscript at various stages of its development are: Jory Burson, Douglas Crockford, Jeff Dyer, Richard Gabriel, Lars T. Hansen, Dave Herman, Pratrap Lakshman, Mathias Bynens, Axel Rauschmayer, Jonathan Sampson, Jon Steinhart, Tom Van Cutsem, Herman Venter, Rick Waldron, Rebecca Wirfs-Brock, and Joseph Yoder.

Richard Gabriel, Rebecca Wirfs-Brock, and Joseph Yoder all participated in exhausting multi-day workshopping sessions where we used comprehensive read-throughs to fine tune the structure and language of the paper.

Memories are fallible, so an accurate history depends on access to primary source documents. The Internet Archive and the Ecma Internationals internal archives provided essential source material for this paper. In particular, this paper could not have been done without the enthusiastic support of Ecma's now-former Secretary General István Sebestyén. Dr. Sebestyén not only ensured that we had access to the private Ecma archives but agreed with us that most of Ecma's document archives relating to TC39 and ECMAScript needed be made publicly accessible via the Web. Ecma's Patrick Charollais assisted in the creation of the https://www.ecma-international.org/archive/ecmascript Web pages.

Finally, Allen Wirfs-Brock wishes to thank Pratrap Lakshman for writing that email in January 2007. It was the beginning of the path that led to this paper.

Fig. 49. Brendan Eich and Allen Wirfs-Brock, 2011. Photo Art courtesy Richard P. Gabriel.

A DRAMATIS PERSONÆ

Name	Affiliation	Role / Contribution
Marc Andreessen	NCSA	Developer of Mosaic Web browser
	Netscape	Cofounder of Netscape; championed Mocha
Jeremy Ashkenas		Created CoffeeScript programming language
Ihab Awad	Google	Contributed to CommonJS module system design
Jan van den Beld	Ecma	Secretary-General 1992–2007
Tim Berners-Lee	CERN, W3G	Inventor of World Wide Web
Eric Bina	NCSA	Developer of Mosaic Web browser
	Netscape	Cofounder of Netscape
Leo Balter	Bocoup	Led Test262 development after ES6 completion
Norris Boyd	Netscape	Early SpiderMonkey developer
Robert Cailliau	CERN	HTML scripting advocate; JavaScript critic
Carl Cargill	Netscape	Standardista; TC39 Vice-Chair 1997
Jim Clark	Netscape	Founder of Silicon Graphics and Netscape
Andrew Clinick	Microsoft	TC39 Vice-Chair 1998–1999; proposed conditional compilation
Donna Converse	Netscape	Contributed to the first draft ES1 specification
Mike Cowlishaw	IBM	Project Editor ES2&ES3; decimal arithmetic proponent
Douglas Crockford	Yahoo!, PayPal	JavaScript evangelist, $ES4_2$ resister; "discovered" JSON
Kevin Dangoor	Khan Academy	Started the ServerJS/CommonJS project
Ryan Dahl	Joyent	Original developer of Node.js
Chris Dollin	HP	Spice language designer
Patrick Dussud	Microsoft	Digital sanitation engineer; JScript garbage collector
Jeff Dyer	Nombas/self	ES4 proponent; ES3&$ES4_1$ constributor
	Macromedia/Adobe	Developed ActionScript 3 compiler; initial $ES4_2$ Editor
Brendan Eich	Netscape	Designer and implementer of original JavaScript
	Mozilla	Mozilla CTO; restarted ES4 effort; Harmony contributor
Cormac Flanagan	Univ Cal Santa Cruz	Hybrid type systems expert; $ES4_2$ design team
David Fugate	Microsoft	Led the initial ES5 phase of Test262 development
Richard P. Gabriel	Sun	Lisper, poet; wrote ECMA-262 Language Overview section
Andreas Gal	Mozilla	Added TraceMonkey optimizer to SpiderMonkey
Michael Gardner	Borland	Co-Editor for first ES1 working draft
Jonathan Gay	Macromedia	Created Flash
Bill Gibbons	Netscape	ES3 working drafts editor
Richard Gillam	IBM	I18N Working Group chair (1998–2000)
Gary Grossman	Macromedia/Adobe	ActionScript designer
Peter Hallam	Google	Traceur transpiler developer
Lars Thomas Hansen	Opera, Adobe	$ES4_2$ editor
Dave Herman	Northeastern Univ	PhD candidate, PL semanticist; $ES4_2$ design team
	Mozilla	Champion of many ES6 proposals including modules
Graydon Hoare	Mozilla	$ES4_2$ design team
Luke Hoban	Microsoft	Led Microsoft TC39 delegation starting in 2011
Waldemar Horwat	Netscape	JS2/$ES4_1$ designer and editor
	Google	Contributor to ES5&ES6
Chris Houck	Netscape	Second member JavaScript team; named SpiderMonkey
Mr. Huffadone	Callscan	Excused from attending the first TC39 meeting
Oliver Hunt	Apple	ES5&ES6 contributor
Scott Isaacs	Microsoft	DHTML developer, MS Live framework architect
Bill Joy	Sun	Hacker, cofounder of Sun; executive sponsor of JavaScript work with Netscape
Yahuda Katz	jQuery Fdn/Tilde	Became a module champion; influenced ES6 classes design

Name	Affiliation	Role / Contribution
Shon Katzenberger	Microsoft	Developed many pseudocode algorithms for ES1
Kris Kowal		CommonJS Module system designer
Peter Kukol	Microsoft	Wrote JScript parser
Pratap Lakshman	Microsoft	$ES4_2$ resister; ES5 Editor; originated ES5conform test suite
Russell Leggett	<es-discuss>	Suggested a "safety syntax" for classes
Norbert Lindenberg	Mozilla	Editor of ECMA-402 1st edition
Julia Liuson	Microsoft	Visual Basic GM; Wirfs-Brock's boss
Steve Leach	HP	Spice language designer
Clayton Lewis	Netscape	First manager of Netscape JavaScript team
David McAllister	Adobe	Standardista; Ecma CC member 2008–2011
Tom McFarland	HP	Internationalization expert
Sam McKelvie	Microsoft	Early JScript interpreter developer
Sebastian McKenzie	Emmerich Manual HS	High school student; developed Babel transpiler
C. Rand McKinny	Netscape	Technical writer assigned to 1st JavaScript specification
Mark S. Miller	Google	OCAP expert; ES5&ES6 contributor; Proxy co-champion
Neil Mix	<es-discuss>	Suggested new ES5 property attribute names
Anup Murarka	Spyglass	Tentatively appointed Assistant Editor at 1st TC39 meeting
John Neumann	Ecma	Standardista; TC39 Chair 2008–2015
Anh Nguyen	Netscape	Represented Netscape at 1st TC39 meeting
Brent Noorda	Nombas	Developed an embedded systems ECMAScript dialect
Bob Nystrom	Google	Traceur transpiler developer
Jason Orendorff	Mozilla	Prototyped module loader for ES6
Adam Peller	IBM	ES5 contributor
Dave Raggett	HP/W3C	Spice originator
Thomas Reardon	Microsoft	Leader of Internet Explorer development
Sam Ruby	IBM	ES5&ES6 contributor; prototyped decimal arithmetic
Alex Russell	Dojo Foundation	ES5 contributor
	Google	Google Chrome Web standards lead; ES6 contributor
William Schulze	Macromedia	TC39-TG1 Convener 2004–2005
István Sebestyén	Ecma	Secretary-General 2007–2019
Dan Smith	Macromedia/Adobe	Director of Engineering Flash Runtime
Edwin Smith	Macromedia/Adobe	Developed AVM2 virtual machine for ActionScript 2&3
Walter Smith	Microsoft	Apple NewtonScript veteran; JScript spec all-nighter
Randy Solton	Borland	Co-editor for 1st ES1 working draft
Maciej Stachkowiak	Apple	Safari/WebKit developer
Guy L. Steele Jr.	Sun	The original Schemer; ES1 Project Editor
David Stryker	Netscape	VP of Core Technologies
Brian Terlson	Microsoft	Led Test262 development for ES6; ECMA-262 editor starting July 2015
Lee Thornason	Macromedia	Developed JVM-hosted Flash prototype
Sam Tobin-Hochstadt	Northeastern Univ	ES6 modules co-champion
Jim Tressa	Oracle	Introduced TC39 to component models
Isabelle Valet-Harper	Microsoft	Standardista; Ecma Co-Ordinating committee member
Tom Van Cutsen	Vrije Universiteit	ES6 contributor; Proxy co-champion
Herman Venter	Microsoft	$ES3\&ES4_1$ contributor; JScript.net developer
Richard Wagner	NetObjects	Started JavaScript Components project within TC39
Rafael Weinstein	Google	Proposed four-stage TC39 development process
Rick Waldron	jQuery Fdn/Bocoup	Systematized TC39 note taking; Editor ECMA-402, 2nd edition
Robert Welland	Microsoft	First JScript developer; JScript spec all-nighter
Chris Wilson	Microsoft	IE Platform Architect

Name	Affiliation	Role / Contribution
Scott Wiltamuth	Microsoft	TC39 Vice-Chair; ES1 Technical WG rapporteur
Allen Wirfs-Brock	Microsoft	$ES4_2$ resister; ES5/5.1 Editor
	Mozilla	ES6 Editor
Rok Yu	Microsoft	JScript program manager, TC39-TG1 Convener 2004
Alon Zakai	Mozilla	Developed Emscripten
Boris Zbarsky	Mozilla	Engineer for DOM bindings and browser standards
Kris Zyp	Dojo Foundation	ES5 contributor

B DRAMATIS CORPORATIONES

Name	Role
Adobe	Graphic tools software company. Acquired Macromedia in 2005.
America Online (AOL)	Acquired Netscape Communications Corp in 1998.
Apple	Computer and mobile device manufacturer. Developed Safari browser.
Borland International	A leading developer of software development tools and compilers.
CERN	Physics reseaerch center where World Wide Web was initially created.
Dojo Foundation	A non-profit that promoted the open source Dojo Toolkit.
Ecma International	Swiss-based computing-related standards organization.
General Magic	Developer of an innovative but commercially unsuccessful software platform for mobile devices.
Google	Internet search and advertising behemoth. Developed Chrome browser.
Hewlett-Packard (HP)	Very large manufacturer of PCs, workstations, servers, and printers.
IEFT	Internet Engineering Task Force, creates Internet standards.
IBM	Very large software services and legacy mainframe computers company.
ISO/IEC	International Standards Organization.
Joyent	Initial corporate host for Node.js development.
Macromedia	Developer of Flash. Acquired by Adobe 2006.
Microsoft Corporation	The dominant personal computer systems and applications software company.
MicroUnity	In early 1990s, a well-funded startup developing video media processors.
Mozilla Foundation	Open-source developer of Firefox browser. Spin-off from Netscape.
NCSA	National Center for Supercomputing Applications at UIUC.
Nombas	Developed scripting languages. Supported embedded JavaScript.
Netscape Communications Corp	Developer of the Netscape Browser, acquired by AOL in 1998.
Object Management Group(OMG)	Consortium founded to develop standards for distributed object systems.
Opera Software	Developer of the Opera familiy of Web browsers.
PayPal	Developed an electronic payment system. eBay subsidiary 2002–2014.
Silicon Graphics Inc. (SGI)	High-performance graphic workstation manufacturer.
Spyglass	Illinois software company licensed by NCSA to commercialize Mosaic.
Sun Microsystems Inc.	Computer manufacturer. Developer of Java. Acquired by Oracle in 2009.
SunSoft	A division of Sun Microsystems.
TC39	The Ecma International technical committee responsible for JavaScript standardization.
UIUC	University of Illinois at Urbana-Champaign.
Yahoo!	Developer of an early widely used Web portal and search engine.
WHATWG	Web Hypertext Application Technology Working Group, an ad hoc group developing HTML-related standards.
W3C	World Wide Web Consortium, Tim Berners-Lee led organization that creates WWW standards.

C GLOSSARY

ActionScript n. the ECMAScript dialect that is the programming language of Adobe Flash.
attribute n. 1. in the ECMAScript specification, a configurable characteristic of a property. 2. in HTML, a behavioral modifier within an opening tag.
AWK n. a domain-specific text processing language [Aho et al. 1988] originally for Unix.
binding n. an association mapping a name to a variable or to a constant value.
breaking-change n. a change to a programming language or platform that causes existing programs to be rejected or to malfunction.
browser wars n. periods of intense competition among browser vendors for market dominance.
Chrome n. a Web browser developed and distributed by Google.
Chromium n. the open-source core of the Chrome browser.
class n. a programming language concept corresponding to a mechanism for defining the common interface and implementation shared by a group of objects.
classical inheritance n. an inheritance mechanism whereby objects acquire their state and behavior from chains of class definitions.
CoffeeScript n. a programming language created by Jeremy Ashkenas that compiles to JavaScript.
CommonJS n. a project started by Kevin Dangoor to develop JavaScript technologies for non-browser environments.
compiler n. an engine that translates a program into (typically) machine language for direct execution by a processor.
constructor n. cf constructor function.
constructor function n. a JavaScript function that allocates and initializes an object and which may be invoked using the `new` operator.
cyclic garbage collection v. a memory management process that is able to reclaim the space allocated to isolated circular structures.
Dart n. a class-based object-oriented programming language developed by Google with the original intent of supplanting JavaScript in Web browsers.
declarative adj. a computational approach based on describing the characteristics of a desired result.
delegation n. a mechanism whereby an object may acquire some or all of its state and behavior from other objects rather than from class definitions.
discriminated union n. a data record with multiple alternative internal structures wherein the actual structure is indicated via an explicit tag value.
destructuring n. referring to the properties of an array or object using syntax similar to array or object literals.
desugar v. to decompose a programming language statement or operation into more fundamental statements and operations.
DevDiv n. Microsoft's Developer Tools Division.
dynamic language n. a programming language that requires little or no analysis of programs prior to execution; most language-mandated error-checking occurs during execution, and typically programs may be constructed or modified as they execute; cf. *static language*.
dynamically typed adj. a programming language where enforcement of data-type–safety constraints is primarily performed during program execution.
es-discuss n. public email forum for discussing ECMAScript evolution.
ECMA-262 n. the ECMAScript Language Specification.
ECMA-402 n. the ECMAScript Initialization API Specification.
engine n. a mechanism for executing a program.
es4-discuss n. original name of es-discuss email forum.
ES.next n. sometimes used within TC39 to refer to the next version of ECMA-262.
exotic object n. a JavaScript object lacking the default behavior for one or more of the essential internal methods that must be supported by all objects; cf. *ordinary object*.
expando property n. a property that is dynamically added to an object after its creation.

factory function n. a function that returns a new object.
Firefox n. a Web browser developed and distributed by Mozilla.
first-class adj. a programming language runtime entity that can be used as a data value; for example, assigned to variables, used as function arguments, returned from functions, or stored in data structures.
Flash n. Adobe's multimedia software platform supporting Rich Internet Applications and other uses.
free variable n. a binding that is referenced but not defined in the local scope.
function n. a subroutine; a parameterized subpart of a program.
hackathon n. an event where programmers gather for a few days to collaborate on a project.
Harmony n. TC39's code name for ongoing ECMA-262 development after abandoning $ES4_2$.
host object n. an object or Class of objects, provided by a JavaScript engine, which provide access to facilities of a host application or platform .
imperative adj. a computational approach based on describing a sequence of steps to be performed to achieve a desired result.
inherit v. in an object-oriented language, to indirectly acquire characteristics.
inheritance n. in object-oriented languages, a mechanism whereby an object inherits some or all of its data and behavior.
inherited property n. a property of a JavaScript object that is inherited from a prototype.
internal method n. a mechanism not part of the language that defines part of the semantics of an object.
internal property n. an aspect of an object that internal methods use to store state needed to define part of the semantics of the object.
interpreter n. an engine that executes a program by traversing a representation of the program and performing each operation as it is encountered.
internationalization n. the process of enabling a program to process multiple human languages, scripts, and writing conventions.
Internet Explorer n. a Web browser developed and distributed by Microsoft.
Java n. a class-based object-oriented programming language developed by Sun Microsystems.
JavaScript engine n. an implementation of the JavaScript language.
JScript n. a dialect of JavaScript implemented by Microsoft.
lambda expression n. a defined function not bound to an identifier, especially the expression that defines expected arguments and steps of execution or evaluation; from the lambda calculus and Lisp.
leaky abstraction n. an abstraction that unintentionally reveals details about the abstraction that should be hidden or private.
mashup n. a Web page that dynamically combines JavaScript code and content from independently managed servers.
membrane n. a mechanism used in object-capability systems for tamper-proof sharing of objects between security contexts.
metaobject protocol n. in an object-oriented language, a well-specified interface for defining and accessing the fundamental language-level behaviors of an object.
method n. a function that is a component part of an object.
Netscape Navigator n. a Web browser developed and distributed by Netscape Communications.
Mocha n. the code name of the language that became JavaScript; also, the name of Netscape's original JavaScript engine.
Mosaic n. a Web browser developed at NCSA by Marc Andreessen and Eric Bina.
Node.js n. a JavaScript-based server platform initially developed by Ryan Dahl in 2009.
nominal type system n. a type system where each type definition introduces a unique data type; in some object-oriented languages a class definition is treated as a nominal type definition.
non-normative adj. a portion of a standards document that does not define any requirements.
no-op n. an operation that does nothing.
normative adj. a portions of a standards document that defines requirements.
object n. a computational device that groups data and behavior into a first-class composite entity; the mechanics of defining and manipulating objects varies among programming languages.
Opera n. a Web browser developed and distributed by Opera Software.

own property n. a property of a JavaScript object that is an integral part of the object rather than inherited.

ordinary object n. a JavaScript object having the default behavior for the essential internal methods that must be supported by all objects; cf. *exotic object*.

profile n. a set of capabilities tailored to the requirements of specific devices, platforms, or applications.

polyfill n. a library that provides APIs that should be provided by a browser but which are missing.

proper tail call n. a tail call that never returns control to the calling function.

property n. a component part of a JavaScript object.

property key n. a string or symbol used to identify a specific property of an object.

prototype n. an object that provides inherited state and behavior to other objects.

prototypal inheritance n. an inheritance mechanism whereby an object acquires some or all of its state and behavior from chains of prototypes.

Safari n. a Web Browser developed by Apple.

sandbox n. a mechanism for isolating a program or a portion of a program so that it can not directly access data from or interfere with the host environment and other programs.

Secure ECMAScript n. an ECMAScript dialect that removes features that could be used by security exploits.

self-hosting n. implementing portions of a programming language engine using code written in that same language.

shadow v. to override but not redefine an inherited characteristic.

Silverlight n. a Microsoft platform for Rich Internet Applications.

scope n. the region of a program in which a variable (or any declared binding) may be referenced.

scope contour n. the representation of a single scope within a group of nested scopes.

scripting language n. a typically simple programming language for orchestrating the operations of a computing system, applications, or the computational abstractions defined using other languages.

SpiderMonkey n. the JavaScript engine used by all Netscape and Mozilla browsers developed after 1996.

static language n. a programming language that requires some or substantial analysis of programs prior to execution; most language-mandated error-checking occurs prior to execution, and typically programs may not be modified as they execute; cf. *dynamic language*.

statically typed adj. a programming language where enforcement of data-type–safety constraints is primarily performed prior to program execution.

tail call n. in a method, a method or function invocation that is the final action of the method; the implementation of such calls may, but need not, return of control to the calling method; cf. *proper tail call*.

transpiler n. a language processor that compiles a program in one language into a program in another language.

type n. a category of values whose elements share common characteristics such as representation and available operations.

type annotation n. a syntactic form used to associate a type with a variable or other binding.

URL n. the address of a World Wide Web page (Uniform Resource Locator).

value n. a unit of information manipulated by programs; in a typed programming language values are categorized into various types.

V8 n. the JavaScript engine used by the Chrome browser.

WebKit n. the open-source browser core used by Apple Safari and some other browsers.

Web 2.0 n. a Web application style that focuses on user-generated content; often highly interactive and built using AJAX techniques.

Web Reality n. the technical aspects and characteristics of the World Wide Web as it exists and is used by existing Web pages and applications; extensions to Web infrastructure typically must allow those existing aspects and characteristics to remain as they are.

D ABBREVIATIONS AND ACRONYMS

AJAX	Asynchronous JavaScript and XML
API	Application Program Interface
CC	Co-Ordinating Committee—Ecma International's management committee
CLR	Common Language Runtime component of Microsoft .NET
CSP	Content Security Policy
CSS	Cascading Style Sheets
DHTML	Dynamic HTML
DOM	Document Object Model
GA	General Assembly—Semi-annual meeting of all Ecma members
GCC	GNU C Compiler
GWT	Google Web Toolkit
HTML	Hypertext Markup Language
IE	Microsoft's Internet Explorer browser
IIFE	Immediately Invoked Function Expression
I18N	Internationalization (18 letters between "I" and "n")
JIT	Just In Time compiler
JVM	Java Virtual Machine
MOP	Metaobject Protocol
NCSA	National Center for Supercomputing Applications
OCAP	Object Capability
OMG	Object Management Group
POSIX	Portable Operating System Interface modeled on UNIX
RFP	Request For Proposal
SES	Secure ECMAScript
TC	A Technical Committee of Ecma International
TDZ	Temporal Dead Zone
TG	A Task Group within an Ecma International TC
VM	Virtual Machine
WG	A Working Group—an ad hoc group within an Ecma TC or TG

E TIMELINES

E.1 Timeline for Part 1: The Origins of JavaScript

1989		At CERN, Tim Berners-Lee begins working on a hypertext project
1990	December 25	Tim Berners-Lee's first Web browser is completed
1991	August 6	Tim Berners-Lee's public announcement of the "WorldWideWeb"
1992		Brendan Eich leaves Silicon Graphics, joins MicroUnity
	December	Marc Andreessen and Eric Bina start development of NCSA Mosaic browser
1993	June	Alpha release of Mosaic for Unix
	November 11	Mosaic released for Microsoft Windows
1994	April	Jim Clark and Marc Andreessen found company eventually named Netscape Communications Corporation
	September	First public beta of Netscape's browser
	December	Production release of Netscape's browser (Navigator 1.0)
1995	Q1	Brendan Eich recruited by Netscape to "do Scheme in the browser"
	April	Alpha release of Java 1.0 by Sun
	April 3	Brendan Eich starts work at Netscape, assigned to server team
	April–May	Strategizing and debate within Netscape about a browser scripting language
	May 6–15(?)	Brendan Eich's 10-day sprint to implement "Mocha"
	May 16(?)	Demo of Mocha to Netscape engineering staff
	May 16–31(?)	Decision made to include Mocha in Netscape 2.0
	May 23	Sun announces Java, and Netscape announces licensing of Java for inclusion in its browser
	May 26	Bill Gates distributes his "Internet Tidal Wave" memo
	Summer	Eich works on Mocha/browser integration, JavaScript features, and bugs
	August 9	Netscape IPO
	August 16	Microsoft Internet Explorer 1.0 released
		Netscape 2.0 feature freeze
	September 18	Netscape 2.0 beta 1 released, includes both LiveScript and Java 1.0
	October	Robert Welland joins Microsoft "to put scripting into Internet Explorer"
	Q4	At Microsoft, Robert Welland, Sam McKelvie, Peer Kukol are reverse engineering beta LiveScript and incubating an interpreter to run it
	November 22	Microsoft Internet Explorer 2.0 released for Windows 95 and NT
	December 4	Netscape/Sun press release announcing JavaScript
	December 5	Microsoft says it intends for Visual Basic to be a standard for Web applications
1996	Q1	Shon Katzenberger takes over JScript interpreter and adds VBScript support
1996	March 18	Netscape 2.0 with JavaScript 1.0 ships
		Netscape LiveWire Server with JavaScript 1.0 ships
	May 29	Internet Explorer 3.0 beta 1 released with JScript and VBScript
	August 13	Internet Explorer 3.0 with JScript 1.0 ships
	August 19	Netscape 3.0 with JavaScript 1.1 ships
	Q3	Brendan Eich builds SpiderMonkey engine to replace Mocha, starts designing JavaScript 1.2
		Clayton Lewis becomes manager of Netscape JavaScript and grows team
	November 21–22	Ecma TC39 startup meeting
	December	Netscape 4 beta 1 ships with SpiderMonkey and initial JavaScript 1.2 features
		Microsoft ships Active Server Pages (ASP 1.0) with JScript and VBScript
1997	January	JScript 2.0 update for Internet Explorer 3 ships
	April	Netscape 4 beta 3 ships, adds regular expression support to JavaScript 1.2
	June	Netscape 4.0 ships with JavaScript 1.2
	September	First edition of the ECMAScript standard is completed and released
	October	Internet Explorer 4.0 ships with JScript 3.0

E.2 Timeline for Part 2: Creating a Standard

1995	December 4	Netscape and Sun announce intent to propose JavaScript to the W3C and IETF as an open standard
1996	March	Netscape 2.0 with JavaScript 1.0 ships
	Q1–Q2	Informal contacts between Netscape and Ecma Secretary-General
	August	Netscape 3.0 with JavaScript 1.1 ships
		Internet Explorer 3.0 with JScript 1.0 ships
	Q3	Brendan Eich starts implementing JavaScript 1.2
	September	Ecma Co-Ordinating Committee approves JavaScript standard startup meeting
	October 10	Netscape applies for Ecma Associate Membership
	October 30	Ecma issues open invitation to a "start-up meeting on a project on Java Script"
	November 21–23	Ecma TC39 startup meeting, "ECMAScript" is placeholder name
		Netscape and Microsoft contribute preliminary JavaScript specifications
		Michael Gardner of Borland appointed interim TC39 editor
	December	Netscape 4 beta 1 ships with SpiderMonkey and initial JavaScript 1.2 features
		Gardner works with Eich, Katzenberger, et al on common 1st-draft specification
	December 18	Microsoft applies for Ecma Ordinary Membership
	December 19–20	Ecma GA creates TC39 as the Web Languages Technical Committee and approves a program of work
1997	January	JScript 2.0 (JavaScript 1.1 features) update for Internet Explorer 3 ships
	January 10	First draft of JavaScript specification distributed to TC39 members
	January 14–15	2nd TC39 meeting, Scott Wiltamuth's list of possible language names
		Technical Working Group (WG) formed with weekly meetings/teleconferences
	January 15	First Technical WG meeting
		Agreed that standard will exclude post JavaScript 1.1 features and host APIs
	January 15–22(?)	Borland will not join Ecma so Gardner withdraws as editor
	January 31	Guy Steele becomes TC39 editor
	Q1	Weekly Technical WG meetings
	March 12	12th draft of specification distributed
	March 18–19	3rd TC39 meeting, Technical WG authorized to complete specification and submit it for GA approval
		Still searching for a name, Netscape teases LiveScript availability
	May 5	18th draft specification submitted to Ecma Secretariat for GA review
	June	Netscape 4.0 ships with JavaScript 1.2
	June 26–27	Ecma General Assembly approves JavaScript standard as *ECMA-262*
		Defers publication until editorial corrections and resolution of naming issue
	July 15–16	4th TC39 meeting. Naming: cannot use LiveScript, name still unresolved
		Preliminary discussions of "Version 2" goals, process, and features
	September 16–17	TC39 agrees on ECMAScript name and releases *ECMA-262* for publication
		Agreement that "V2" would be backward compatible with 1st edition
	September 23–24	Ecma submits *ECMA-262* into the ISO/IEC fast-track process
	October	Internet Explorer 4.0 ships with JScript 3.0—claims *ECMA-262* compliance
		Guy Steele resigns as TC39 editor, replaced by Mike Cowlishaw
		Technical WG starts meeting monthly
	October 9	6 month ISO/IEC ballot period begins
	October 10	Technical WG drafts first feature list for "Version 2"
1998	Q1	Nearly complete turnover of TC39 technical contributors
		Brendan Eich joins project to open source Netscape browser code
	February 18	TC39 meeting sets June 1999 as "V2" publication target

1998	February 19	Technical WG meeting
		Brendan Eich, last recorded attendance; first attendance:
		Waldemar Horwat (Mozilla), Herman Venter (Microsoft), Rok Yu (Microsoft)
		Exception handling proposals from Netscape and Microsoft
	March 31	Netscape open sources browser source code at mozilla.org
	April 9	ISO/IEC ballot ends, 27 pages of comments submitted
	April 22	First draft "V2" specification, based on ES1 specification
	May	Bill Gibbons becomes "V2" working draft editor
		Technical WG starts using "Status Document" to track progress
		HP submits comments about I18N support requirements
	May 18	US Department of Justice files browser related antitrust suit against Microsoft
	June 15	ISO Disposition of Comments meeting resolves all ballot issues
	July	Updated specification submitted to ISO for publication as ISO/IEC 16262:1998
	August	Ecma publishes *ECMA-262 2nd Edition*
		Netscape 4.06 ships with JavaScript 1.3—claims *ECMA-262* compliance
	September	ES2 changes merged into "V2" working draft
	September 16	I18N WG established
	November 18	I18N meeting: Richard Gillam of IBM to chair I18N WG
		Only minimal localization hooks planned for next ECMA-262 specification
		Majority of I18N functionality should be in separate library/specification
	November 19	Technical WG meeting
		IBM proposes including decimal arithmetic
		"Futures List" reviewed and updated, many items deferred until post "V2"
		Global binding for `undefined` added
		Concern that browsers may ship different exception hierarchies before "V2"
	November 28	AOL announces agreement to acquire Netscape Communications Corp
1999	Q1	Much work on nested functions and closures
		Scoping differences between Netscape and Microsoft implementations
		Regular Expressions: should they include Perl 5 features?
		TC39 shifts to referring to next release as "E3" instead of "V2"
	February 19	Waldemar Horwat reveals JavaScript 2.0 design
	March 17	AOL completes acquisition of Netscape
	March 29	TC39 meeting: slips schedule by 6 months. New target December 1999
	March 30	Technical WG does another triage of "E3" features list
	Q2	Intensive work to resolve issues and finish specification
	July 12–13	Technical WG detailed section-by-section review of working draft
	August 8	E3 Status document shows "content agreed" or unchanged for all clauses
	August 20	Bill Gibbons completes "Edition 3 Final Draft Candidate"
	September 23–24	Final Technical WG "E3" meeting, Bill Gibbons has left for new job
		Herman Venter and Waldemar Horwat will complete specification
		"Joined functions" added to specification
		Final agreement on exception classes
		Agreement on scoping of the identifier in a named *FunctionExpression*
	September 24	TC39 votes to refer ECMA-262, 3rd Edition to Ecma GA
	October 13	Final Draft sent to Ecma Secretariat
	November 15–16	Minor corrections applied to Final Draft
		Microsoft discovered that `String.replace`, as specified, breaks websites; final draft changed to match Microsoft's previous behavior
	December 16–17	Ecma General Assembly approves *ECMA-262, 3rd Edition*
2000	March 25	Waldemar Horwat creates a public ES3 Errata Web page
	July	Microsoft ships IE 5.5 with ES3 compliant JScript 5.5
	November	Netscape ships Netscape 6 with ES3 compliant JavaScript 1.5

E.3 Timeline for Part 3: Failed Reformations

1997	July	Oracle presentation on ECMAScript as a component scripting language
1998	Q1	Waldemar Horwat and Herman Venter start participating on TC39
	February	Dave Raggett's Spice proposal submitted to TC39
	May 4–5	Dave Raggett cochairs W3C Shaping the Future of HTML Workshop
		W3C decides to freeze HTML and "make a fresh start with the next generation of HTML based upon a suite of XML tag-sets"
	May 15	Jeff Dyer's first TC39 meeting attendance
	June	NetObjects 1st draft of ECMAScript Components specification
	November	Dave Raggett presents revised Spice proposal to TC39
		TC39 interested in classes, numeric units, tuples, and modules
		TC39 Spice Working Group established
	December	Spice WG teleconference followed by new Raggett proposal
1999	Q1	Spice WG discussions, disagreements about static versus dynamic approaches
	February 19	Waldemar Horwat reveals Netscape JavaScript 2.0 specification
	March	Spice WG becomes Modularity Subgroup
	March 30	TC39 creates a "Futures List" for Edition 4 and beyond
	May	Macromedia Flash ships with simple JavaScript-like scripting language
	June	*ECMA-290 ECMAScript Components Specification* approved by Ecma GA
	Q4	TC39's attention turns to $ES4_1$
	October 14	Modularity Subgroup meeting at HP Labs, Bristol England
	November	TC39 drafts $ES4_1$ provisional features list
	December 16–17	Ecma General Assembly approves *ECMA-262, 3rd Edition*
2000	January	Microsoft wants "Edition 4" finished by December, wants to cut features
		Microsoft circulates proposed ES3 specification changes for type annotations
	June 22	Microsoft announces the .NET framework
	July 11	Microsoft distributes .NET preview including early version of JScript.NET
	July 13	Herman Venter discusses JScript.NET design at TC39 meeting
	August	Macromedia Flash 5 ships with ActionScript—an ECMAScript dialect
	August 17	$ES4_1$ Netscape proposal split from JavaScript 2.0 proposal
	August 22	Herman Venter and Waldemar Horwat meet and try to align JavaScript 2 and JScript.net; discuss 43 points of disagreement
2001		Douglas Crockford starts evangelizing JavaScript
	January	Scope of TC39 expanded to include .NET; ECMAScript work moves to TC39-TG1
	June	*ECMA-327 ECMAScript 3rd Edition Compact Profile* Approved by Ecma GA
	August 27	Internet Explorer 6 released
	November 17	Waldemar Horwat JS2.0 paper at MIT Lightweight Languages Workshop
2002	March	Target date for $ES4_1$ completion moved to December 2003
	June	BEA proposes project to add XML extensions to ECMAScript (E4X)
		Herman Venter attends his last TC39-TG1 meeting
	August	Work starts on E4X specification
	September 22	Phoenix 0.1 (proto-Firefox) browser released
	December	Douglas Crockford introduces the world to JSON, sets up json.org website
2003	January	Apple announces WebKit and Safari Browser
	March	TG1 discusses postponing $ES4_1$ to focus on E4X, but doesn't do it
	May	Jeff Dyer joins Macromedia, works on developing ActionScript 3
	July 15	AOL shuts down Netscape operations, lays off most of staff
		Waldemar Horwat resigns as $ES4_1$ editor
		TG1 suspends work on $ES4_1$ to focus on E4X
		Independent Mozilla Foundation launched
	September	Macromedia releases ActionScript 2, loosely based on $ES4_1$ syntax

2003	November	Macromedia joins Ecma to participate in TC39-TG1
	December	*ECMA-357 ECMAScript for XML* specification approved by Ecma GA
2004	May	Mozilla Foundation joins Ecma as a Not For Profit member
	June	Brendan Eich helps organize WHATWG
		Brendan Eich attends his first TC39 (TG1) meeting since February 1998
		TG1 decides to abandon $ES4_1$ specification for "something less complex"
		Jeff Dyer appointed ES4 editor
	Q3–Q4	TG1 primarily working on E4X 2nd edition
	November 9	Firefox 1.0 browser released
2005		Dojo framework released
	Q1–Q3	TG1 primarily working on E4X 2nd edition
	February	Jesse James Garrett coins the term "AJAX"
	Q2–Q4	Brendan Eich blogging and speaking about $ES4_2$ issues and goals
	April	Adobe announces agreement to acquire Macromedia
	September	Brendan Eich appointed TC39-TG1 convener
		Eich focuses TG1 on developing $ES4_2$
		Schedule targets: feature agreement 6 months, draft in 1 year
		$ES4_2$ specification notation will be an "Operational semantic language"
	September 26	Brendan Eich keynote at ICFP: "JavaScript at Ten Years"
	October	Brendan Eich in blog post, interested in a formal "checkable specification"
	November	Macromedia contributes ActionScript 3 specification to TG1
		Graydon Hoare's first TG1 meeting, representing Mozilla
	November 30	Firefox 1.5 with JavaScript 1.6 ships
	December	*ECMA-357 2nd Edition E4X* approved by Ecma GA
		Adobe completes acquisition of Macromedia
2006		JQuery and MooTools frameworks released
		TG1 collects and discusses $ES4_2$ proposals using private wiki
	January	Adobe contributes AS3-derived draft "Ecmascript 4 Language Specification"
	February	Lars Hansen's first TG1 meeting, representing Opera
		Dave Herman's first TG1 meeting
		Dave Herman TG1 presentation on formal specification techniques
		Maciej Stachowiakof's first TG1 meeting, representing Apple
	March	Cormac Flanagan's first TG1 meeting
	Q2-Q3	Dave Herman explores various available formal specification languages
	April	Pratap Lakshman's first TG1 meeting, representing Microsoft
		TG1 target is $ES4_2$ validated and ready for GA submission, end Q1 2007
	June	Adobe announces Flash ActionScript 3
		Public ES4 wik export and es4-discuss mailing list publicly accessible
		Douglas Crockford's first TG1 meeting, representing Yahoo!
		Crockford raises ES3 compatibility concerns regarding $ES4_2$
		Pratap Lakshman says Microsoft intends to implement ES4 after IE7
	July 27–28	TG1 meeting, Douglas Crockford stresses backward compatibility importance
		Pratap Lakshman says backward compatibility is Microsoft's highest priority
	October 18	Internet Explorer 7 released
	October 19–20	TG1 decides to use ML-based reference implementation to specify $ES4_2$
	remainder Q4	Initial work on reference implementation using SML
	October 24	Firefox 2.0 with JavaScript 1.7 released, includes various Python and $ES4_2$ inspired experimental features
	November 6	Tamarin: Adobe contributes of AS3VM to Mozilla (open source)
	November 15	Douglas Crockford holds a Browser Security Summit at Yahoo!
	November 16	Mike Cowlishaw attends TG1, IBM wants decimal arithmetic in $ES4_2$

2006	December	Jeff Dyer experiments with integrating ML code with specification text
		TG1 $ES4_2$ WG holds weekly teleconference and monthly meetings/hackathons
		Ongoing work on hybrid structural type system and other new semantics
		Ongoing work on building $ES4_2$ ML Reference Implementation
2007	January	Pratap Lakshman emails Microsoft DevDiv managers asking direction on $ES4_2$
		Allen Wirfs-Brock recommends that Microsoft oppose $ES4_2$
	February	TG1 meeting, Pratap Lakshman announces Microsoft opposition to $ES4_2$ effort
		Wirfs-Brock & Crockford agree to develop joint proposal for ES3 maintenance
	March 15	Joint Microsoft/Yahoo! proposal to refocus TG1 on maintaining ES3
	March 21–23	TG1 meeting at Microsoft, Allen Wirfs-Brock's first TG1 meeting
		Contentious discussions about $ES4_2$ effort and TG1 goals
		Agreement that Microsoft and Yahoo! could start ES3.1 project
	March 29	Crockford & Wirfs-Brock meet, draft ES3.1 Goals and Design Principles
	April	Lars Hansen goes to work for Adobe, continues to work on $ES4_2$
	April 4	Douglas Crockford updates his recommended modifications to ECMAScript
	April 15	Initial ES3.1 wiki proposal largely derived from Crockford recommendations
	April 18–20	TG1 meeting, ES3.1 WG resists basing 3.1 on ML reference implementation
		Adam Peller's first TG1 meeting, representing IBM
	Summer	Pratap Lakshman analyzes JavaScript interoperability between major browsers
	June 8	Announcement of public "M0" release of the $ES4_2$ Reference Implementation
	June 21	Alex Russell's first TG1 meeting, representing Dojo Foundation
	June 22	Call-to-action to start $ES4_2$ specification writing, for completion by September
	Q3–Q4	Public disputes regarding ES3.1 and $ES4_2$ in blog posts and conference panels
	September 7	$ES4_2$ specification completion target reset to September 2008
		Lars Hansen appointed $ES4_2$ editor
	September 26	Pratap Lakshman releases "JScript Deviations from ES3" report
	September 27–28	TG1 meeting to winnow $ES4_2$ wiki proposals
		Approves 54 proposals, 26 proposals rejected or deferred
		Jeff Dyer pushes to reject "Maintenance of ES3" proposal
		Decision to remove ES3.1 as $ES4_2$ proposal and give it its own wiki namespace
		Waldemar Horwat's first TG1 meeting since 2003 (guest of convener)
	October 16	Google becomes Ecma Ordinary Member, Waldemar Horwat represents Google
	October 21	Lars Hansen completes 40 page "ECMAScript 4 Language Overview"
	October 23–24	Concerns about discord within TC39-TG1 raised at Ecma CC meeting
	November	Lars Hansen releases three reports on various aspects of $ES4_2$
	November 8–9	TG1 meeting attended by Ecma President John Neumann
		Neumann recommends elevating TG1 to full TC status; more Ecma supervision
		Douglas Crockford proposes new "Secure ECMAScript (SES)" project
		Straw poll shows significant TG1 interest in ES3.1, $ES4_2$, and SES
	December	Adobe and Microsoft agree to co-sponsor John Neumann as new TC39 chair
2008		TC39-TG1 is again simply TC39, other former TC39 TGs transferred to TC49
	Q1-Q2	ES3.1 WG gets organized and working on new specification derived from ES3
	February	Jeff Dyer publishes a new $ES4_4$ work plan
		Dyer & Hansen propose deferring "strange, unproven, or costly" $ES4_2$ features
	May	Douglas Crockford publishes his book: *JavaScript: The Good Parts*
	May 29–30	TC39 meeting draft specifications for ES3.1 and $ES4_2$ introduced
	June	Adobe withdraws from $ES4_2$ project
	June 17	Firefox 3.0 with JavaScript 1.8 released
	July 23–25	TC39 meeting in Oslo ends $ES4_2$. TC39 to focus on ES3.1 and "Harmony"
	August	Public announcement of termination of $ES4_2$ and start of Harmony project
2009	December 3	*ECMA-290 ECMAScript Components* withdrawn as an Ecma standard
2015	June 17	*ECMA-327 Compact Profile* and *Ecma-357 E4X* withdrawn as Ecma standards

E.4 Timeline for Part 4: Modernizing JavaScript

2006	May	Google releases GWT (Java to JavaScript transpiler)
2007	December	Apple releases SunSpider JavaScript benchmark suite
	October	Google Caja Project (secure JavaScript) publicly announced
2008	January 24	Mark Miller's first TC39 meeting, representing Google
		Kris Zyp's first TC39 meeting, representing Dojo Foundation
	February 21	Start of twice-weekly ES3.1 WG design teleconferences
	February 26	Revised ES3.1 goals with 3 out of 4 browser feature adoption rule
	March	Pratap Lakshman is ES3.1 specification editor
		ES3.1 base document created from ES3 specification plus errata
		Writing tasks assigned to seven ES3.1 WG participants
	April 22	es3.1-discuss email forum opens
		Allen Wirfs-Brock posts design sketch for `Object.defineProperty`
	April 24	Strict mode concepts and `"use strict";` directive discussed
	May 29–30	ES3.1 draft specification introduced at TC39 meeting and posted to wiki
	June	Adobe withdraws from $ES4_2$ project
	June 10	Mark Miller updates ES3.1 draft to use structured pseudocode
	June 17	Firefox 3.0 with JavaScript 1.8 includes "expression closures"
	July 4	ES3.1 draft uses lexical environments instead of activation objects
		Block-scoped `const` declarations
	July 15	Allen Wirfs-Brock posts "Static Object Functions: Use Cases and Rationale"
	July 23–25	TC39 meeting in Oslo ends $ES4_2$, new focus is ES3.1 and "Harmony"
		Harmony discussion includes desugaring classes to lexical closures
	August	Public announcement of termination of $ES4_2$ and start of Harmony project
		Harmony Strawman page created on TC39 wiki
	August 28	First meeting of TC39 Secure ECMAScript (SES) WG
	September 1	ES3.1 draft includes initial support for decimal arithmetic
	September 2	Google preview release of Chrome browser with V8 JavaScript engine
	October 13	Waldemar Horwat on es-discuss lists four binding "dead zone" alternatives
		Dave Herman strawman proposal for "Lambdas"
	November	Cormac Flanagan posts first Harmony strawman relating to classes
	November 19–20	TC39 meeting, ES3.1 final feature review
		Decimal arithmetic and `const` declarations deferred to Harmony
	November 21	Wiki Strawman page has 7 entries
	November 29	Brendan Eich proposal for Wirfs-Brock's Smalltalk-like "Block Lambdas"
	December 11	Google Chrome 1.0 released
2009	January	CommonJS project starts
		Kris Kowal and Ihab Awad present CommonJS modules precursor to TC39
		Douglas Crockford launches ADsafe
	January 28	TC39 meeting, Pratap Lakshman demonstrates Microsoft ES3.1 prototype in IE
	March 19	Internet Explorer 8 released with partial support for ES3.1 features
	March 24	Last meeting of SES WG
	March 25–26	Pratap Lakshman resigns as ES3.1 editor, Allen Wirfs-Brock assumes role
		ES3.1 renamed to ES5; "ES4" designation permanently retired
	April 7	"Final Draft" of ES5 specification posted to TC39 wiki
	May	First public version of Node.js
		Eric Arvidsson's first TC39 meeting, representing Google
		Brendan Eich "catchalls" Harmony strawman proposal
	June	Microsoft contributes ES5 new features test suite to Ecma
		Google releases open-source Sputnik ES3 test suite
	June 17	Apple Safari updated with Nitro JavaScript engine

2009	June 24	Firefox 3.5 with TraceMonkey JavaScript optimization released
	July	Harmony Goal Statement posted on TC39 wiki
	August	Harmony Strawman wiki page has lists 21 proposals
	August 17	Apple discovers that ES5 `argument` object changes break websites
	August 27	ES5 Release Candidate 1 posted
	September 23	TC39 votes to forward ES5 to Ecma GA for its approval
	October 28	ECMA-262 5th Edition sent to Ecma GA for review
	December	Jeremy Ashkenas starts developing CoffeeScript
		Tom Van Cutsem es-discuss post: "Catch-all proposal based on proxies"
	November 5	Sam Tobin-Hochstadt's first TC39 meeting, Northeastern University
	November 7	Brendan Eich says Harmony needs a second-class module system
	December 3	Ecma GA approves and publishes *ECMA-262 5th Edition*
2010		Remy Sharp coins the term "polyfill"
		Ben Alman coins the term "IIFE"
	Q1	Dave Herman joins Mozilla
		Dave Herman and Sam Tobin-Hochstadt develop "Simple Modules" design
	January	ISO fast track process started for ES5
		Tom Van Cutsen's first TC39 meeting, Vrije Universiteit
	February	Ihab Awad presents "Emaker Style" module proposal
	April	Feature themes added to Harmony Goals wiki page
	May	TC39 starts Test262 project, combines Microsoft ES5conform & Google Sputnik
		Ihab Awad recommends that TC39 focus on Simple Modules proposal
	September	Alon Zakai releases Emscripten for compiling C code to JavaScript
		TC39 split on whether classes are primarily for "high integrity abstractions"
	December	Allen Wirfs-Brock leaves Microsoft, joins Mozilla
		Harmony Strawman wiki page lists 66 proposals
2011		Four ES6 specification drafts released in 2011
	January	Brendan Eich publishes "Harmony of My Dreams" blog post
	March	Wirfs-Brock proposes extending object literals to support class-like abstractions
		Loose consensus that classes should support construct/prototype/instance triad
		Simplified "Simple Module" proposal presented
	March 14	Internet Explorer 9 released with Chakra JavaScript engine; fully supports ES5
	March 22	Firefox 4.0 and JavaScript 1.8.5 released; fully supports ES5
	May	Brendan Eich alternative proposals: Block Lambda Revival & Arrow Functions
		Google Traceur transpiler project announced
	May 6	Dave Herman demos multi-language Harmony module loader at JSConf
	May 10	Allen Wirfs-Brock, Mark Miller, others meet on joint class proposal
	May 24–26	TC39 Harmony "feature freeze" strawman winnowing
		Classes accepted for Harmony based upon new joint class proposal
		Following meeting wiki shows approximately 45 accepted Harmony proposals
	June	*ECMA-262 Edition 5.1* and identical *ISO/IEC 16262:2011* published
	June 22	Allen Wirfs-Brock clones the ES5.1 specification re-titled as "Draft Edition 6"
	June 27	Dave Herman says Miller's classes too complex, suggests a minimal class design
	July 12	Allen Wirfs-Brock posts first Harmony (ES6) specification working draft
	October	Tom Van Cutsem and Mark Miller evolve Proxies into Direct Proxies
	November 11	Dave Herman posts a Minimal Classes strawman
	December	Dave Herman proposes "ES6 doesn't need opt-in" on es-discuss
2012		Nine ES6 specification drafts released in 2012
	January	Dave Herman "One JavaScript" presentation adopted by TC39
	March 19	Russell Leggett on es-discuss calls for a "safety syntax" for classes
	March 25	Allen Wirfs-Brock inspired by Leggett proposes "Maximally Minimal Classes"
	March 28–29	Brendan Eich guides TC39 to consensus on adopting Arrow Functions

2012	May	Ordinary and exotic object terminology adopted for use in ES6 specification TC39 agrees to allow specification work on Maximally Minimal Classes Yahuda Katz and Rick Waldron attend their first TC39 meeting representing jQuery Foundation Rick Waldron takes meeting notes, begins to systematize capture of technical meeting notes
	Q3–4	Jason Orndorff & Dave Herman prototype Harmony modules+loader in Firefox
	September 27	ES6 specification draft released that includes Maximally Minimal Classes
	October	Microsoft introduces TypeScript transpiler
	December	*ECMA-402, 1st Edition ECMAScript Internationalization API* published
2013		Eight ES6 specification drafts released in 2013 TC39 primarily addressed approved proposals feature and specification issues
	September	Rafael Weinstein & Dmitry Lomov propose a new TC39 development process
	October	*ECMA-404 The JSON Data Interchange Format* published Promises added to ES6 to avoid them being subsumed into HTML specification
	November	Dave Herman posts first drafts of the Realm API
2014		Nine ES6 specification drafts released in 2014 Node.js community criticizing TC39 for not adopting CommonJS module design TC39 starts using new multi-stage process to develop post ES6 features
	January	Preliminary version of module&loader pseudocode in draft ES6 specification
	April	Douglas Crockford's last TC39 meeting
	Summer	Yehuda Katz creates jsmodules.io website introducing ES6 modules to Node.js programmers
	June	Browser developers raise concerns about ES6 class semantics for subclassing built-in constructors
	July	Array and generator comprehensions dropped from draft ES6 specification
	September	Module loader dropped from draft ES6 specification
	September 24	Two competing redesigns presented for the subclassing built-ins problem
	October	Module specification (without loader) completed in draft ES6 specification
2015		Eight ES6 specification drafts released in 2015
	January 27	TC39 reaches final consensus on outstanding ES6 issues including subclassing constructors
	February	Babel (aka 6to5) transpiler introduced
	March	TC39 approves ECMAScript 2015 specification for referral to Ecma GA
	April 14	Final Draft of ES2015 posted to TC39 wiki
	Q2–4	TC39 following new process working on ES2016 and longer term proposals
	June 17	*ECMA-262 6th Edition ECMAScript 2015 Language Specification* and *ECMA-402 2nd Edition ECMAScript Internationalization API* approved as Ecma standards
	July	Brian Terlson succeeds Allen Wirfs-Brock as ECMA-262 project editor
2016	June 14	*ECMA-262 7th Edition ECMAScript 2016 Language Specification* and *ECMA-402 3rd Edition ECMAScript Internationalization API* approved as Ecma standards

F DECEMBER 4, 2005 JAVASCRIPT ANNOUNCEMENT

[Netscape and Sun 1995, Page 1–2]

Netscape and Sun Announce Javascript, the Open, Cross-Platform Object Scripting Language For Enterprise Networks and the Internet

28 Industry Leading Companies to Endorse Javascript as a Complement to Java for Easy Online Application Development

---

MOUNTAIN VIEW, Calif. (December 4, 1995) – Netscape Communications Corporation (NASDAQ: NSCP) and Sun Microsystems, Inc. (NASDAQ:SUNW), today announced JavaScript, an open, cross-platform object scripting language for the creation and customization of applications on enterprise networks and the Internet. The JavaScript language complements Java, Sun's industry-leading object-oriented, cross-platform programming language. The initial version of JavaScript is available now as part of the beta version of Netscape Navigator 2.0, which is currently available for downloading from Netscape's Web site.

In addition, 28 industry-leading companies, including America Online, Inc., Apple Computer, Inc., Architext Software, Attachmate Corporation, AT&T;, Borland International, Brio Technology, Inc., Computer Associates, Inc., Digital Equipment Corporation, Hewlett-Packard Company, Iconovex Corporation, Illustra Information Technologies, Inc., Informix Software, Inc., Intuit, Inc., Macromedia, Metrowerks, Inc., Novell, Inc., Oracle Corporation, Paper Software, Inc., Precept Software, Inc., RAD Technologies, Inc., The Santa Cruz Operation, Inc., Silicon Graphics, Inc., Spider Technologies, Sybase, Inc., Toshiba Corporation, Verity, Inc., and Vermeer Technologies, Inc., have endorsed JavaScript as an open standard object scripting language and intend to provide it in future products. The draft specification of JavaScript, as well as the final draft specification of Java, is planned for publishing and submission to appropriate standards bodies for industry review and comment this month.

JavaScript is an easy-to-use object scripting language designed for creating live online applications that link together objects and resources on both clients and servers. While Java is used by programmers to create new objects and applets, JavaScript is designed for use by HTML page authors and enterprise application developers to dynamically script the behavior of objects running on either the client or the server. JavaScript is analogous to Visual Basic in that it can be used by people with little or no programming experience to quickly construct complex applications. JavaScript's design represents the next generation of software designed specifically for the Internet and is:

- designed for creating network-centric applications
- complementary to and integrated with Java
- complementary to and integrated with HTML
- open and cross-platform.

Java, developed by Sun, is an object-oriented programming language that operates independent of any operating system or microprocessor. Java programs called applets can be transmitted over a network and run on any client, providing the multimedia richness of a CD-ROM over corporate networks and the Internet. Java has been widely hailed by programmers because it eliminates the need to port applications, and by managers of information systems for its potential to lower the costs of distributing and maintaining applications across the network.

With JavaScript, an HTML page might contain an intelligent form that performs loan payment or currency exchange calculations right on the client in response to user input. A multimedia weather forecast applet written in Java can be scripted by JavaScript to display appropriate images and sounds based on the current weather readings in a region. A server-side JavaScript script might pull data out of a relational database and format it in HTML on the fly. A page might contain JavaScript scripts that run on both the client and the server. On the server, the scripts might dynamically compose and format HTML content based on user preferences stored in a relational database, and on the client, the scripts would glue together an assortment of

Java applets and HTML form elements into a live interactive user interface for specifying a net-wide search for information.

Java programs and JavaScript scripts are designed to run on both clients and servers, with JavaScript scripts used to modify the properties and behavior of Java objects, so the range of live online applications that dynamically present information to and interact with users over enterprise networks or the Internet is virtually unlimited. Netscape will support Java and JavaScript in client and server products as well as programming tools and applications to make this vision a reality.

"Programmers have been overwhelmingly enthusiastic about Java because it was designed from the ground up for the Internet. JavaScript is a natural fit, since it's also designed for the Internet and Unicode-based worldwide use," said Bill Joy, co-founder and vice president of research at Sun. "JavaScript will be the most effective method to connect HTML-based content to Java applets."

Netscape's authoring and application development tools – Netscape Navigator Gold 2.0, Netscape LiveWire and Netscape LiveWire Pro – are designed for rapid development and deployment of JavaScript applications. Netscape Navigator Gold 2.0 enables developers to create and edit JavaScript scripts, while Netscape LiveWire enables JavaScript programs to be installed, run and managed on Netscape servers, both within the enterprise and across the Internet. Netscape LiveWire Pro adds support for JavaScript connectivity to high-performance relational databases from Illustra, Informix, Microsoft, Oracle and Sybase. Java and JavaScript support are being built into all Netscape products to provide a unified, front-to-back, client/server/tool environment for building and deploying live online applications.

Java is available to developers free of charge. The Java Compiler and Java Developer's Kit as well as the HotJava browser and related documentation are available from Sun's Web site at http://java.sun.com. In addition, the Java source code can be licensed for a fee. Details on licensing are also available via the java.sun.com Web page. To date, Sun has licensed Java to a number of leading technology companies, including Borland, Macromedia, Mitsubishi, Netscape, Oracle, Silicon Graphics, Spyglass, and Toshiba. Sun's Workshop for Java toolkit is scheduled for release in Spring 1996. Sun's NEO product family, the first complete development, operating and management environment for object-oriented networked applications, will also use Java-enabled browsers as front-ends to the NEO environment.

Netscape and Sun plan to propose JavaScript to the W3 Consortium (W3C) and the Internet Engineering Task Force (IETF) as an open Internet scripting language standard. JavaScript will be an open, freely licensed proposed standard available to the entire Internet community. Existing Sun Java licensees will receive a license to JavaScript. In addition, Sun and Netscape intend to make a source code reference implementation of JavaScript available for royalty-free licensing, further encouraging its adoption as a standard in a wide variety of products.

Netscape Communications Corporation is a premier provider of open software for linking people and information over enterprise networks and the Internet. The company offers a full line of Netscape Navigator clients, Netscape servers, development tools and Netscape Internet Applications to create a complete platform for next-generation, live online applications. Traded on Nasdaq under the symbol "NSCP", Netscape Communications Corporation is based in Mountain View, California.

With annual revenues of $6 billion, Sun Microsystems, Inc. provides solutions that enable customers to build and maintain open network computing environments. Widely recognized as a proponent of open standards, the company is involved in the design, manufacture and sale of products, technologies and services for commercial and technical computing. Sun's SPARC(TM) workstations, multiprocessing servers, SPARC microprocessors, Solaris operating software and ISO-certified service organization each rank No. 1 in the UNIX(R) industry. Founded in 1982, Sun is headquartered in Mountain View, Calif., and employs more than 14,000 people worldwide.

Additional information on Netscape Communications Corporation is available on the Internet at , by sending email to info@netscape.com or by calling 415-528-2555. Additional information on Sun Microsystems is available on the Internet at http://www.sun.com or, for Java information, http://java.sun.com Netscape Communications, the Netscape Communications logo, Netscape, and Netscape Navigator are trademarks of Netscape Communications Corporation. JavaScript and Java are trademarks of Sun Microsystems, Inc. All other product names are trademarks of their respective companies.

G ISSUES LIST FROM FIRST TC39 MEETING

The following list of issues to be resolved for the first version of the standard is from the Minutes of the November 21-22 TC39 organizing meeting [TC39 1996].

1. Unicode support (Feature)
2. Semantics of delete (Semantics)
3. Semantic of indexing (S)
4. Binding contexts: Inheritance properties - scope for "…" - Host object model versus language (S)
5. No unified discussion of a storage model (S)
6. The argument array; semantics implication of the semantics of arrays (argument becomes a keyword); important in recursive functions
7. Arguments are currently very costly - performance related
8. EVAL on every object does not sound like a good idea
9. Caller should be optional
10. Block sharing scope - an issue when program becomes large
11. Implicit Globals are bad. It would be nice to clarify to programmers the differences between local and global variables
12. Object prototypes in user defined constructors. What is the intent of prototype ? It is unclear and needs better definition. Netscape says that this is a bug.
13. Support of CTL Z as a wide space character
14. Is NULL a type or a distinct object reference
15. 1Array length = 0 is illegal
16. Slot vs property
17. "this is" undefined or well-defined in function
18. Whether there is a global object
19. "this" in method call
20. Type and value of &&/|| operators
21. "For/in" loop enumerates properties in well-defined order
22. Run-time & compile time
23. Pointers: where are they defined ?
24. Versioning
25. Why reserve Java keywords ?
26. Identifier redefinition Error or last definition wins ?
27. f.prototype before new f() ?
28. Grammar per constructor not allowed for new f ?
29. Top level evaluation order
30. Can '\0' occur within a string ?
31. Bytes vs characters (a general length-issue)

H INITIAL PROPOSED ECMASCRIPT VERSION 2 NEW FEATURE LIST

On October 1997 the TC39 technical working group met and produced these lists of features that were candidates for inclusion in "Version 2" of the ECMAScript Specification. The minutes of the October meeting are lost. The meeting notes [TC39 1998c] for February 19, 1998, identify the following as propagated and updated from the 1997.10.10 notes.

Agreed items for Version 2

caller (omitted from V1)
do while
break to label
continue to label
switch
regexp
=== operator (strict equality)
conditional compilation
literal notation
function closures (expression, nesting)
reveal __parent__ , __photo__ [sic]
arguments object
exception handling
toSource (people want a way to make objects persistent)
Function.prototype.apply
instanceof

Other Items in consideration for V2

binary object
Date (as presented by Borland in 1997)
generic sequence operations on a string or an array
threading issues
undefined literal, not reserved
parse {int, float} step point result
toString extensions
date to string
toBoolean (object)
Hide proto.property
meta object protocol (MOP)
package concept

I A PARTIAL E3 DRAFT STATUS REPORT

[Cowlishaw 1999b]

ECMA TC39 E3 status ECMA/TC39/99/3

ECMAScript E3 draft status [1999.04.08]

This page lists the sections in the ECMAScript 3rd edition draft, along with their current status.

Dates are dates of last change of status.
Leaf sub-sections unchanged from the 2nd edition are omitted.
Editorial changes do not affect status.
Notes are open questions, reasons for status change, *etc*.
Section numbers are base document numbers, unless shown with '?'.

1	Unchanged since V1
N	Not ready
D	Discussion needed
F	Function accepted
C	Content agreed

Date	Status	#	Heading	Notes
1998.04.20	N	**1**	**Scope**	
1999.03.30	C	**2**	**Conformance**	
1999.03.30	F	**3**	**References**	I18N
-	1	**4**	**Overview**	
-	1	4.2	Language Overview	
-	1	4.3	Definitions	
-	1	**5**	**Notational Conventions**	
-	1	5.1	Syntactic and Lexical Grammars	
1998.05.15	C	5.2	Algorithm conventions	
1999.03.30	F	**6**	**Source Text**	I18N
1999.03.30	D	**7**	**Lexical Conventions**	'atoi' problem [WH]
1999.03.30	F	7.1	White Space	I18N
1999.03.30	C	7.2	Line Terminators	
1998.09.15	D	7.4	Tokens	Regexp
1998.04.20	N	7.4.2	Keywords	[WG] some from 7.4.2
1998.04.20	N	7.4.3	Future Reserved Words	[WG] some move to 7.4.1
1999.03.30	F	7.5	Identifiers	I18N
1998.05.15	C	7.6	Punctuators	
1998.09.15	C	7.7	Literals	
1999.03.30	C	7.7.4	String Literals	
1999.02.19	F	7.7.5	Regular Expression Literals	regexp
1998.06.03	N	7.8	Automatic semicolon insertion	[WH] Grammatical ambiguities
1998.06.03	N	7.8.1	Rules	[WH]

1 of 6 08.04.99 08:53

J JANUARY 11, 1999 CONSENSUS ON MODULARITY FUTURES

At the January 11–12, 1999, TC39 working groups meeting there was a long discussion of the goals and technical challenges for supporting "programming in the large" in a future edition of ECMAScript. A record of the following points of agreement were recorded in the meeting notes [Raggett 1999b].

Agreed Goals

- Robust libraries and programming in the large
- No interference between libraries/different units of code
- Reasonable efficiency - precompilation feasible
- Continuity with existing ECMAScript and its users (don't break existing code)
- Audience - suits naive users and experienced scripters e.g. a fairly heavy duty forms validation package.
- Language extensibility
- Ease of use

Classes

- Inheritance (at least single)
- A class is a type
- Class syntax declares fields, methods, inheritance
- Slot access syntax is same as zero argument method call syntax
- Herman would call slots properties (decide to call them fields)
- A method is not a slot that contains a function
- Encapsulation (private, public, package scope etc.)
- Expando
- "new" constructors for instances
- "implements"

Interfaces

- Syntax for defining interfaces similar to classes
- Methods includng getters and setters
- Interfaces are types
- exposing one function as another in an interface

Types

- primitive types
- describe constraints on local vars, fields, args and results
- "any" type
- type annotations optional

Packages & Namespaces/Versions

- A syntax for defining packages
- Namespace control - hide things inside a package
- Qualify identifiers by their source packages
- Imports

Stuff we want to discuss but haven't yet agreed on

- Syntax for everything
- Events?
- Expando turn on/off default

- *Member scoping rules
- Multiple inheritance?
- Can interfaces include fields/slots?
- Interface inheritance
- Interface member default
- *Static type-dependent name lookup
- Syntax for type annotations
- *Casting: assert or coerce?
- Operator overloading?
- Method overloading
- Declared types: assert or coerce?
- How many kinds of namespaces are there?
- *Is there a way to robustly add methods, slots
- *and globals to a previously released package?
- Can packages have versions?
- *Global variables and their interaction with packages
- Language versioning (e.g. this package uses style rules)
- Package Interfaces

* These items were identified as show stopper issues

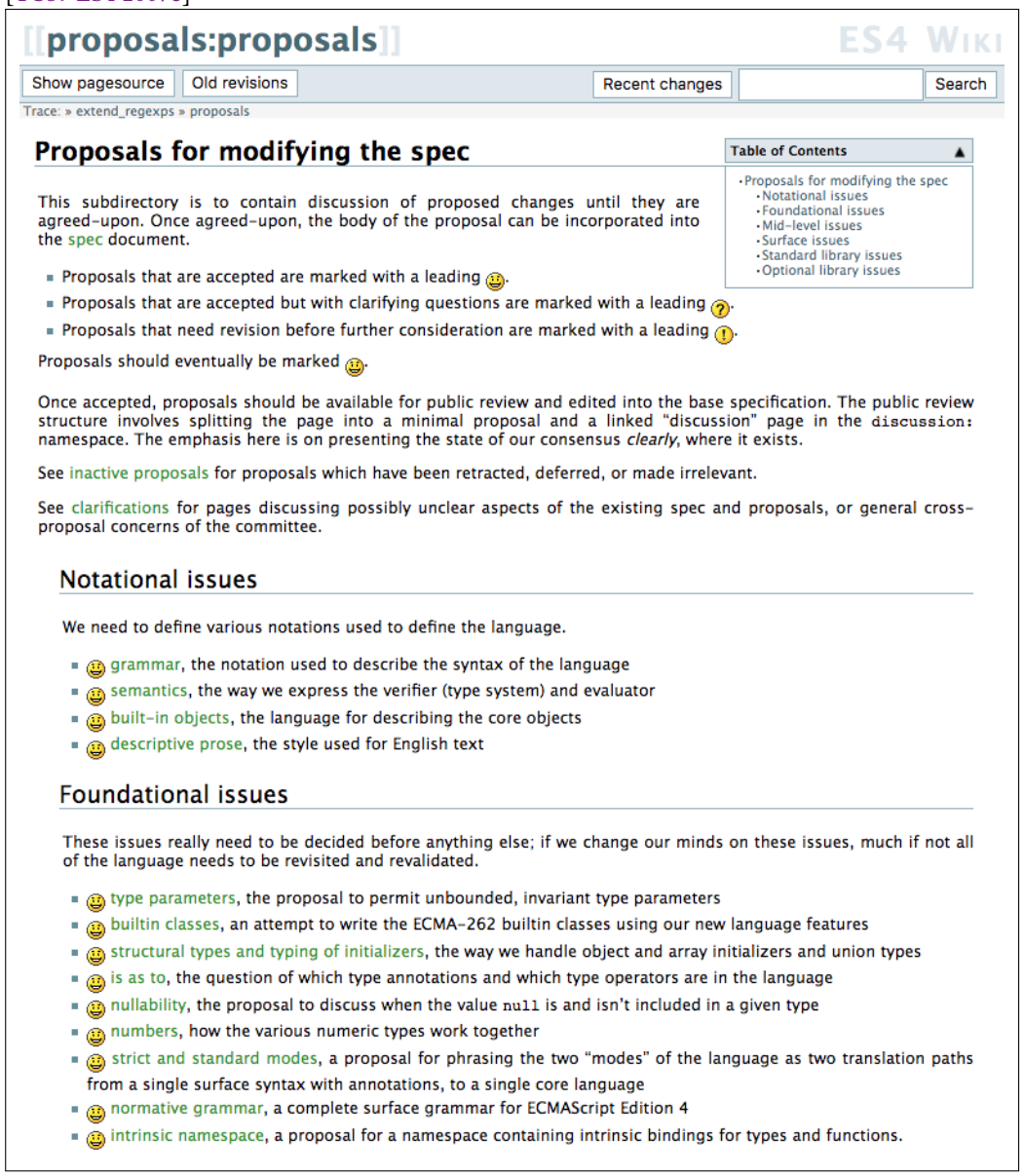
K ES4 REFERENCE IMPLEMENTATION ANNOUNCEMENT

The following announcement was posted by Dave Herman [2007] to the Lambda the Ultimate weblog on June 8, 2007:

> **ECMAScript Edition 4 Reference Implementation**
>
> ECMAScript specification and reference implementation. You can download source and binary forms of the reference implementation.
>
> As we've discussed before here on LtU, the reference implementation of ECMAScript is being written in Standard ML. This choice should have many benefits, including:
>
> - to make the specification more precise than previous pseudocode conventions
> - to give implementors an executable framework to test against
> - to provide an opportunity to find bugs in the spec early
> - to spark interest and spur feedback from the research and user communities
> - to provide fodder for interesting program analyses to prove properties of the language (like various notions of type soundness)
> - to use as a test-bed for interesting extensions to the language
>
> This pre-release is just our first milestone, i.e., the first of many "early and often" releases. Neither the specification nor the reference implementation is complete, and this early implementation has plenty of bugs. We encourage anyone interested to browse the bug database and report additional bugs.
>
> We're happy to hear your feedback, whether it's bug reports or comments here on LtU or on the es4-discuss mailing list.

L ES4$_2$ APPROVED PROPOSALS SEPTEMBER 2007

[TC39 ES4 2007e]



[[proposals:proposals]] ES4 Wiki

Show pagesource | Old revisions | Recent changes | Search

Trace: » extend_regexps » proposals

Proposals for modifying the spec

Table of Contents

- Proposals for modifying the spec
  - Notational issues
  - Foundational issues
  - Mid-level issues
  - Surface issues
  - Standard library issues
  - Optional library issues

This subdirectory is to contain discussion of proposed changes until they are agreed-upon. Once agreed-upon, the body of the proposal can be incorporated into the spec document.

- Proposals that are accepted are marked with a leading .
- Proposals that are accepted but with clarifying questions are marked with a leading .
- Proposals that need revision before further consideration are marked with a leading .

Proposals should eventually be marked .

Once accepted, proposals should be available for public review and edited into the base specification. The public review structure involves splitting the page into a minimal proposal and a linked "discussion" page in the `discussion:` namespace. The emphasis here is on presenting the state of our consensus *clearly*, where it exists.

See inactive proposals for proposals which have been retracted, deferred, or made irrelevant.

See clarifications for pages discussing possibly unclear aspects of the existing spec and proposals, or general cross-proposal concerns of the committee.

Notational issues

We need to define various notations used to define the language.

- grammar, the notation used to describe the syntax of the language
- semantics, the way we express the verifier (type system) and evaluator
- built-in objects, the language for describing the core objects
- descriptive prose, the style used for English text

Foundational issues

These issues really need to be decided before anything else; if we change our minds on these issues, much if not all of the language needs to be revisited and revalidated.

- type parameters, the proposal to permit unbounded, invariant type parameters
- builtin classes, an attempt to write the ECMA-262 builtin classes using our new language features
- structural types and typing of initializers, the way we handle object and array initializers and union types
- is as to, the question of which type annotations and which type operators are in the language
- nullability, the proposal to discuss when the value `null` is and isn't included in a given type
- numbers, how the various numeric types work together
- strict and standard modes, a proposal for phrasing the two "modes" of the language as two translation paths from a single surface syntax with annotations, to a single core language
- normative grammar, a complete surface grammar for ECMAScript Edition 4
- intrinsic namespace, a proposal for a namespace containing intrinsic bindings for types and functions.

Mid-level issues

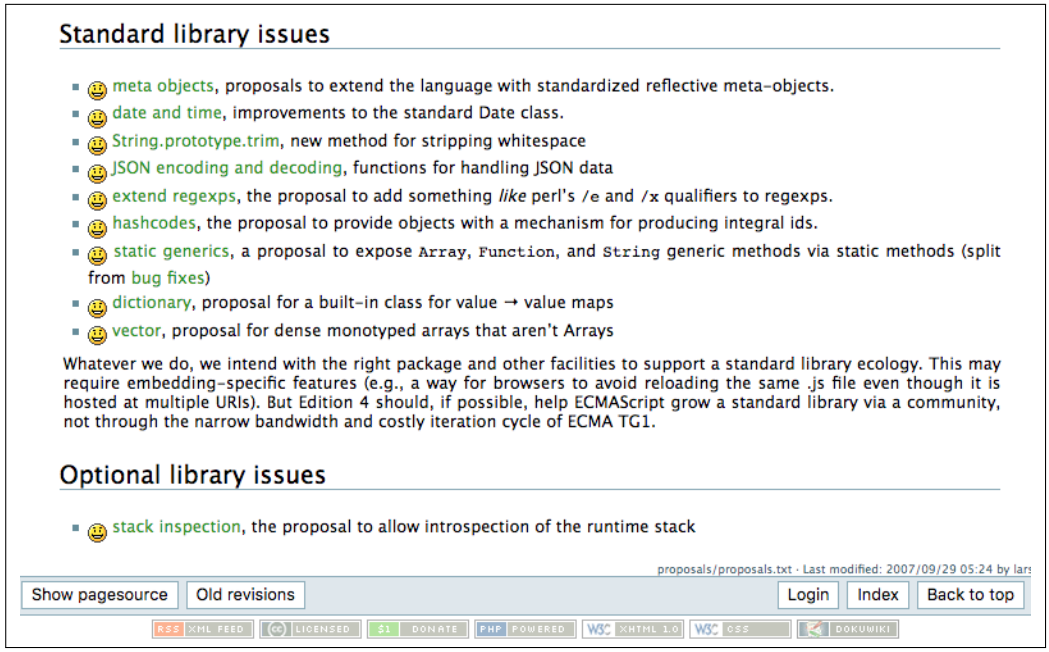
These have broad but reasonably orthogonal implications across the language.

- enumerability, the proposal to define enumerability as a single `DontEnum` bit set by `Object.p.propertyIsEnumerable(name, flag)`
- switch class, the proposal to permit type unions and safe type switching.
- block expressions, the proposal to add a lexically scoped expression form that is not hoisted and that can serve as a "better var".
- proper tail calls, the proposal to enforce bounded control behavior for function calls in tail position
- type definitions, syntax and semantics.
- syntax for type expressions, what is it.
- local packages, a lightweight `private package` form to ease web programming
- namespace shadowing, reference ambiguities are resolved by the block order of `use namespace` and `import` pragma
- iterators and generators, the proposal to copy python's iterator and generator system.
- name objects, which follow from existing `for-in` iteration semantics combined with namespaces.
- expression closures, a proposal for function syntax where the body is an expression.
- remove the arguments object, a proposal to deprecate and replace `arguments` by better mechanisms
- resurrected eval, somewhat misnamed, a proposal to reform `eval` by limiting indirect calls and (in strict mode) binding effects
- arrays, gathering various attempts to make array objects more array-like and therefore useful
- program units, a proposal to allow declaring compilation unit dependencies
- generic functions, a facility for multimethods with type dispatch

Surface issues

These are (possibly) easy to change at the "last minute" without really affecting much else.

- getters and setters, the proposal to specify named get, set, and method initialisers within object initialisers.
- catchalls, the proposal to specify "catchall" get, set, and method-call accessors.
- destructuring assignment, a proposal for syntax to pick apart arrays and structures in assignment and binding forms.
- reformed with, the proposal to extend `with` statement syntax to support precise object typing and lexical scope.
- decimal, the proposal to add IEEE754r-style decimal floating point to the language.
- typeof, what is the result of `typeof` when used with some new fundamental types?
- syntax for pragmas, what is it.
- reserved words, a proposal
- update unicode, the proposal to incorporate unicode v4 escapes (and possibly more) as lexemes.
- bug fixes, a proposal to make a few small, "easy" fixes to ECMA-262 Edition 3.
- globals, the proposal for providing access to the global environment as an object.
- triple quotes, the proposal for multiline string literals.
- slice syntax, a proposal to support slice and splice syntax, for both conciseness and early binding optimization
- line terminator normalization, a proposal to interpret CRLF and CR in source text as LF for purposes of multiline strings and regexps, as well as determining line numbers
- versioning, a proposal to detect the supported version of ECMAScript



Standard library issues

- meta objects, proposals to extend the language with standardized reflective meta-objects.
- date and time, improvements to the standard Date class.
- String.prototype.trim, new method for stripping whitespace
- JSON encoding and decoding, functions for handling JSON data
- extend regexps, the proposal to add something *like* perl's `/e` and `/x` qualifiers to regexps.
- hashcodes, the proposal to provide objects with a mechanism for producing integral ids.
- static generics, a proposal to expose `Array`, `Function`, and `String` generic methods via static methods (split from bug fixes)
- dictionary, proposal for a built-in class for value → value maps
- vector, proposal for dense monotyped arrays that aren't Arrays

Whatever we do, we intend with the right package and other facilities to support a standard library ecology. This may require embedding-specific features (e.g., a way for browsers to avoid reloading the same .js file even though it is hosted at multiple URIs). But Edition 4 should, if possible, help ECMAScript grow a standard library via a community, not through the narrow bandwidth and costly iteration cycle of ECMA TG1.

Optional library issues

- stack inspection, the proposal to allow introspection of the runtime stack

M ECMASCRIPT HARMONY ANNOUNCEMENT

[Eich 2008b]

```
From brendan at mozilla.org  Wed Aug 13 14:26:56 2008
From: brendan at mozilla.org (Brendan Eich)
Date: Wed, 13 Aug 2008 14:26:56 -0700
Subject: ECMAScript Harmony
Message-ID: <C4FE90A0-2762-4061-872B-3E5F174AEACE@mozilla.org>

It's no secret that the JavaScript standards body, Ecma's Technical
Committee 39, has been split for over a year, with some members
favoring ES4, a major fourth edition to ECMA-262, and others
advocating ES3.1 based on the existing ECMA-262 Edition 3 (ES3)
specification. Now, I'm happy to report, the split is over.

The Ecma TC39 meeting in Oslo at the end of July was very productive,
and if we keep working together, it will be seen as seminal when we
look back in a couple of years. Before this meeting, I worked with
John Neumann, TC39 chair, and ES3.1 and ES4 principals, especially
Lars Hansen (Adobe), Mark Miller (Google), and Allen Wirfs-Brock
(Microsoft), to unify the committee around shared values and a common
roadmap. This message is my attempt to announce the main result of
the meeting, which I've labeled "Harmony".

Executive Summary

The committee has resolved in favor of these tasks and conclusions:

1. Focus work on ES3.1 with full collaboration of all parties, and
target two interoperable implementations by early next year.
2. Collaborate on the next step beyond ES3.1, which will include
syntactic extensions but which will be more modest than ES4 in both
semantic and syntactic innovation.
3. Some ES4 proposals have been deemed unsound for the Web, and are
off the table for good: packages, namespaces and early binding. This
conclusion is key to Harmony.
4. Other goals and ideas from ES4 are being rephrased to keep
consensus in the committee; these include a notion of classes based
on existing ES3 concepts combined with proposed ES3.1 extensions.

Detailed Statement

A split committee is good for no one and nothing, least of all any
language specs that might come out of it. Harmony was my proposal
based on this premise, but it also required (at least on the part of
key ES4 folks) intentionally dropping namespaces.

This is good news for everyone, both those who favor smaller changes
to the language and those who advocate ongoing evolution that
requires new syntax if not new semantics. It does mean that some of
the ideas going back to the first ES4 proposals in 1999, implemented
variously in JScript.NET and ActionScript, won't make it into any ES
```

standard. But the benefit is collaboration on unified successor specifications to follow ES3, starting with ES3.1 and continuing after it with larger changes and improved specification techniques.

One of the use-cases for namespaces in ES4 was early binding (use namespace intrinsic), both for performance and for programmer comprehension -- no chance of runtime name binding disagreeing with any earlier binding. But early binding in any dynamic code loading scenario like the web requires a prioritization or reservation mechanism to avoid early versus late binding conflicts.

Plus, as some JS implementors have noted with concern, multiple open namespaces impose runtime cost unless an implementation works significantly harder.

For these reasons, namespaces and early binding (like packages before them, this past April) must go. This is final, they are not even a future possibility. To achieve harmony, we have to focus not only on nearer term improvements -- on "what's in" or what could be in -- we must also strive to agree on what's out.

Once namespaces and early binding are out, classes can desugar to lambda-coding + Object.freeze and friends from ES3.1. There's no need for new runtime semantics to model what we talked about in Oslo as a harmonized class proposal (I will publish wiki pages shortly to show what was discussed).

We talked about desugaring classes in some detail in Oslo. During these exchanges, we discussed several separable issues, including classes, inheritance, like patterns, and type annotations. I'll avoid writing more here, except to note that there were clear axes of disagreement and agreement, grounds for hope that the committee could reach consensus on some of these ideas, and general preference for starting with the simplest proposals and keeping consensus as we go.

We may add runtime helpers if lambda-coding is too obscure for the main audience of the spec, namely implementors who aim to achieve interoperation, but who may not be lambda-coding gurus. But we will try to avoid extending the runtime semantic model of the 3.1 spec, as a discipline to guard against complexity.

One possible semantic addition to fill a notorious gap in the language, which I sketched with able help from Mark Miller: a way to generate new Name objects that do not equate as property identifiers to any string. I also showed some sugar, but that is secondary at this point. Many were in favor of this new Name object idea.

There remain challenges, in particular getting off of the untestable and increasingly unwieldy ES1-3.x spec formalism. I heard some generally agree, and no one demur, about the ES4 approach of using an SML + self-hosted built-ins reference implementation (RI).

We are going to look into stripping the RI of namespaces and early binding (which it uses to ensure normative self-hosted behavior, not susceptible to "user code" modifying the meaning of built-ins), simplifying it to implement ES3.1plus or minus (self-hosted built-ins may require a bit more magic). More on that effort soon.

ES3.1 standardizes getters and setters that were first implemented at Mozilla and copied by Apple and Opera. More such de-facto standardization is on the table for a successor edition in the harmonized committee.

I heard good agreement on low-hanging "de-facto standard" fruit, particularly let as the new var, to match block-scoped const as still proposed (IIRC) in 3.1. Also some favorable comments about simple desugarings such as expression closures and destructuring assignment, and other changes in JS1.7 and 1.8 that do not require new runtime semantic models.

Obviously, these require new syntax, which is appropriate for a major post-3.1 "ES-harmony" edition. Syntax is user interface, there's no reason to avoid improving it. What's more, the intersection semantics of extended ES3 implementations conflict and choke off backward-compatible *semantics* for syntax that may even parse in all top four browsers (e.g., functions in blocks).

Both the appropriateness of new syntax, and the need to make incompatible (with ES3 extensions) semantic changes, motivate opt-in versioning of harmonized successor edition. I believe that past concerns about opt-in versioning requiring server file suffix to MIME type mapping maintenance were assuaged (browsers in practice, and HTML5 + RFC 4329, do not consider server-sent Content-Type -- the web page author can write version parameters directly in script tag type attributes).

Some expressed interest in an in-language pragma to select version; this would require immediate version change during parsing. It's a topic for future discussions.

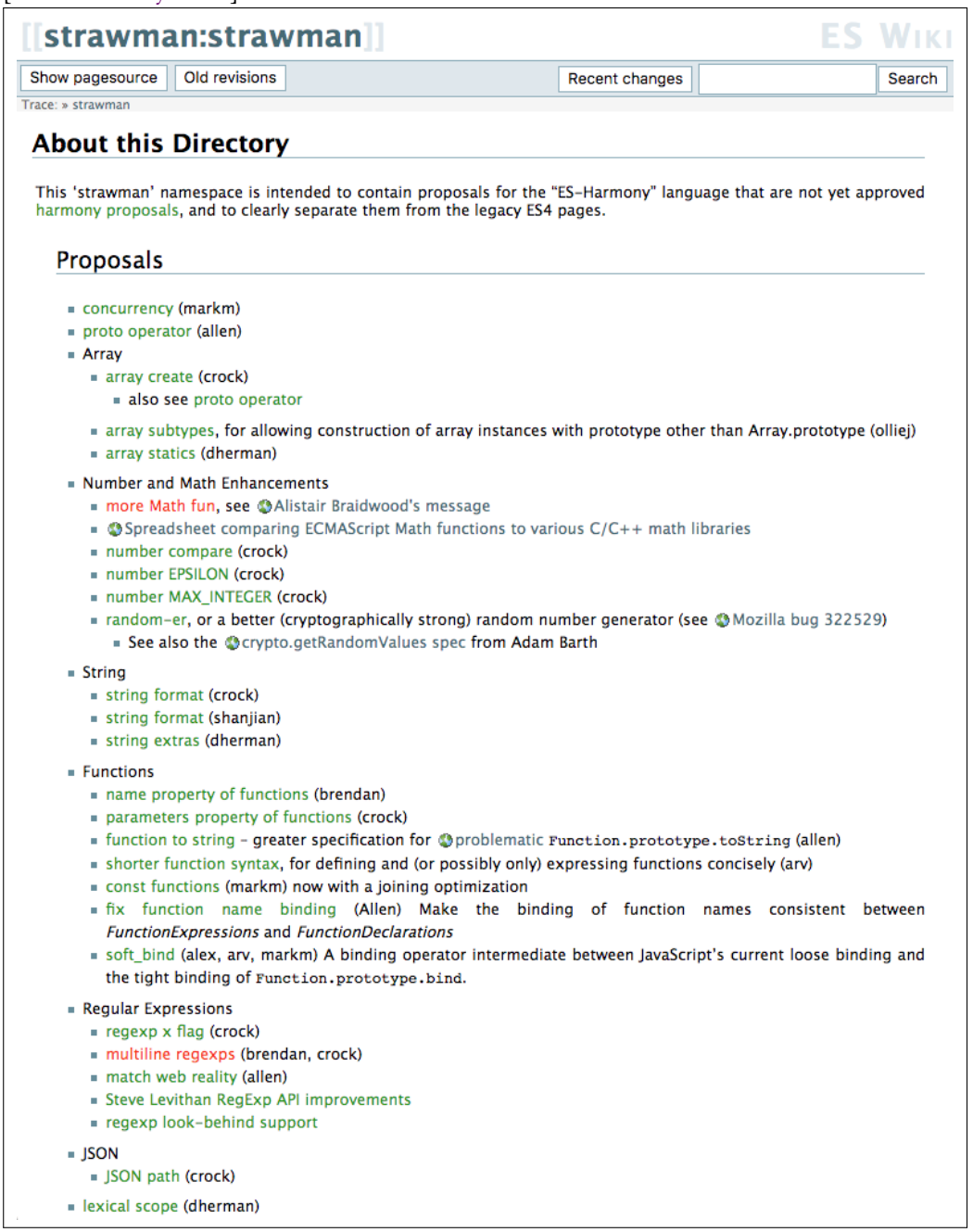
The main point, as important as cutting namespaces in my view, is that the committee has a vision for extending the language syntactically, not trying to fit new semantics entirely within some combination of existing "three of four top browsers" syntax and standard library extensions.

As Waldemar Horwat (Google) said on the final day, the meeting was seminal, and one of the most productive in a long while. Much work remains on 3.1 and Harmony, but we are now on a good footing to make progress as a single committee.

/be

N HARMONY STRAWMAN PROPOSALS MAY 2011

[TC39 Harmony 2011c]



[[strawman:strawman]] ES Wiki

Show pagesource | Old revisions | Recent changes | Search

Trace: » strawman

About this Directory

This 'strawman' namespace is intended to contain proposals for the "ES-Harmony" language that are not yet approved harmony proposals, and to clearly separate them from the legacy ES4 pages.

Proposals

- concurrency (markm)
- proto operator (allen)
- Array
  - array create (crock)
    - also see proto operator
  - array subtypes, for allowing construction of array instances with prototype other than Array.prototype (olliej)
  - array statics (dherman)
- Number and Math Enhancements
  - more Math fun, see Alistair Braidwood's message
  - Spreadsheet comparing ECMAScript Math functions to various C/C++ math libraries
  - number compare (crock)
  - number EPSILON (crock)
  - number MAX_INTEGER (crock)
  - random-er, or a better (cryptographically strong) random number generator (see Mozilla bug 322529)
    - See also the crypto.getRandomValues spec from Adam Barth
- String
  - string format (crock)
  - string format (shanjian)
  - string extras (dherman)
- Functions
  - name property of functions (brendan)
  - parameters property of functions (crock)
  - function to string - greater specification for problematic `Function.prototype.toString` (allen)
  - shorter function syntax, for defining and (or possibly only) expressing functions concisely (arv)
  - const functions (markm) now with a joining optimization
  - fix function name binding (Allen) Make the binding of function names consistent between *FunctionExpressions* and *FunctionDeclarations*
  - soft_bind (alex, arv, markm) A binding operator intermediate between JavaScript's current loose binding and the tight binding of `Function.prototype.bind`.
- Regular Expressions
  - regexp x flag (crock)
  - multiline regexps (brendan, crock)
  - match web reality (allen)
  - Steve Levithan RegExp API improvements
  - regexp look-behind support
- JSON
  - JSON path (crock)
- lexical scope (dherman)

- Modules
  - simple modules (with examples) and module_loaders (dave, samth)
  - simple module functions (markm)
  - (modules_harmonic, (ihab_awad) temporary placeholder)
  - modules_primordials
  - modules_emaker_style
    - system, a place to put powerful objects provided by the embedding without having to introduce names into the global scope every time.
  - modules_packages
- Operators
  - default operator (crock)
  - modulo operator (crock)
  - has operator (crock)
- Value types
  - value types, requirements for first-class number-like objects with operators and (we hope) literal syntax (brendan)
  - value proxies, extending proxies to implement value types (cormac)
- Garbage collection
  - gc semantics Thoughts on specifying the semantics of garbage collection. (markm)
  - weak references and post-mortem finalization. (markm)
  - proper tail calls (markm)
  - inherited explicit soft fields (markm) - an encapsulation-respecting alternative to private names above. See comparison at names vs soft fields.
- Data structures
  - binary data, convenient, high-level, structured binary data (dherman)
    - binary data semantics
    - binary data discussion
  - typed arrays, similar to the above byte arrays, as originally defined by webgl (arun)
  - encapsulated hashcodes (allen - see also weak maps)
  - simple maps and sets (markm)
  - records (brendan, dherman)
  - tuples (brendan, dherman)
  - dicts (dherman)
- Loops, iteration, enumeration
  - iterator conveniences, an API for convenient construction of iterators (dherman)
  - enumeration, more fully-specified semantics for property enumeration in `for-in` loops (brendan, dherman)
  - array comprehensions, convenient, expressive syntax for creating eagerly-computed arrays (brendan, dherman)
- Asynchronous Programming
  - deferred_functions allow writing asynchronous code in a linear style where you would otherwise use callbacks and manual CPS (peterhal)
- Private Names prividing unique, unforgable property names(allen)
  - Instance Variables work in conjunction with Private Names to provide strong encapsulation/information hiding.
  - Also see discussion links at end of Private Names page.
- Declarative Abstractions Based on Object Literals (allen)
  - Declarative Object and Class Abstractions Based Upon Extended Object Initialisers including:
    - Object Initialiser Meta Properties
    - Method Properties
    - Other Object Initialiser Property Attribute Modifiers
      - see proto operator and concise object literal extensions for an alternative to the above 3 proposals
    - Object Initialiser Initialization Blocks
    - Implicit property initialization expressions (brendan)
    - Class Initialisers
      - super in Object Initialisers

- The Private Names extension integrates with extended Object Initialisers:
  - Private Names in Object Initialisers
- Also see destructuring for "record extension and row capture" via `...` - this may need its own proposal page

- Similar declarative object extension mechanisms
  - scoped object extensions (peterhal)
- High Integrity Factories
  - classes with trait composition (markm)
    - traits semantics (tom)
  - guards (markm) A syntax and parameterizable semantics for dynamic type-like checks.
- Versions and Configuration
  - versioning, the full versioning of script tag content and whole-frame/window object model monty (brendan)
  - pragmas, a future-proof "use" directive for language modes and implementation options (dherman)
- quasis, quasi-literals provide DSL support for content generation and introspection (mikesamuel)
- ast, a parser built into conforming Harmony implementations which returns a standard abstract syntax tree (dherman)
- extended Object API, standardizing some "missing" ES5 methods on Object. (tomvc)
- proxy extensions, not-yet-harmonized extensions to proxies. (tomvc)
  - Deferred: proxy instanceof, enabling proxies to trap `instanceof` directly. (tomvc, markm)
  - function proxy prototype, enabling function proxies to specify a custom prototype object (dherman)
  - proxy derived traps, derived `getPropertyDescriptor` and `getPropertyNames` traps (tomvc)
  - handler access to proxy, giving Proxy handlers access to the intercepted proxy (tomvc)
  - proxy set trap, remove a discrepancy between the default set trap behavior of Proxies vs regular ES5 objects (tomvc)
  - derived traps forwarding handler, semantics of the derived traps of the Proxy default forwarding handler (tomvc)
- paren free, relaxing the rules about mandatory parentheses (brendan, dherman)
- completion reform, to make "completion position" a statically predictable attribute (dherman)
- completion let, a variation on `let` expressions that uses completions (dherman)
- debugger expressions, to extend the syntax of `debugger` to be an expression (dherman)
- multiple globals, bringing the spec in line with the reality of multiple global objects (dherman)
- Conditionals
  - catch guards, for conditionally catching exceptions (dherman)
  - pattern matching, a conditional form based on destructuring (dherman)
  - cond expressions, for concise linear nesting of conditional expressions (dherman)
- Specification Techniques
  - specification language, using Harmony to describe Harmony.
  - Another alternative, translating the ES5 spec. into a JavaScript-based definitional interpreter es5definterp.js.txt
- Internationalization support
  - Support full Unicode in strings: remove all 16-bit Unicode assumptions concerning strings and source code
  - Unicode support, tools to aid in using Unicode.
  - i18n API, internationalization API (locales, sorting, formatting, parsing).
- ECMASript object model and internal metaobject protocol (Allen)
  - ES5 internal methods and aligning them with Proxy handlers
  - ES5 internal nominal typing and generalizing usage of [[Class]]

Deferred proposals

See the deferred proposals page.

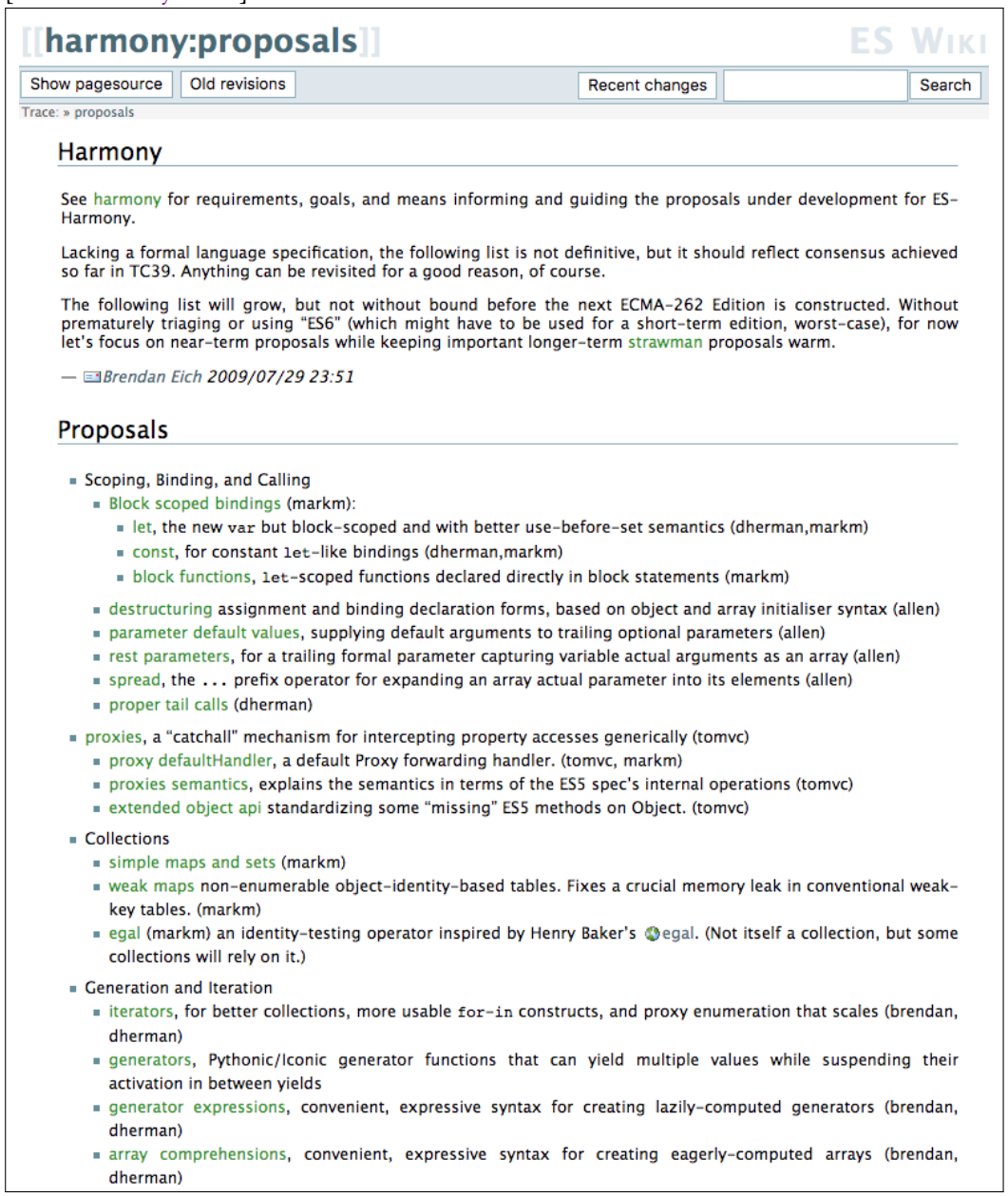
strawman/strawman.txt · Last modified: 2011/04/28 17:01 by allen

Show pagesource | Old revisions | Login | Index | Back to top

RSS XML FEED | CC LICENSED | $1 DONATE | PHP POWERED | W3C XHTML 1.0 | W3C CSS | DOKUWIKI

O HARMONY PROPOSALS WIKI PAGE FOLLOWING MAY 2011 TRIAGE

[TC39 Harmony 2011b]



[[harmony:proposals]] ES Wiki

Show pagesource | Old revisions | Recent changes | Search

Trace: » proposals

Harmony

See harmony for requirements, goals, and means informing and guiding the proposals under development for ES-Harmony.

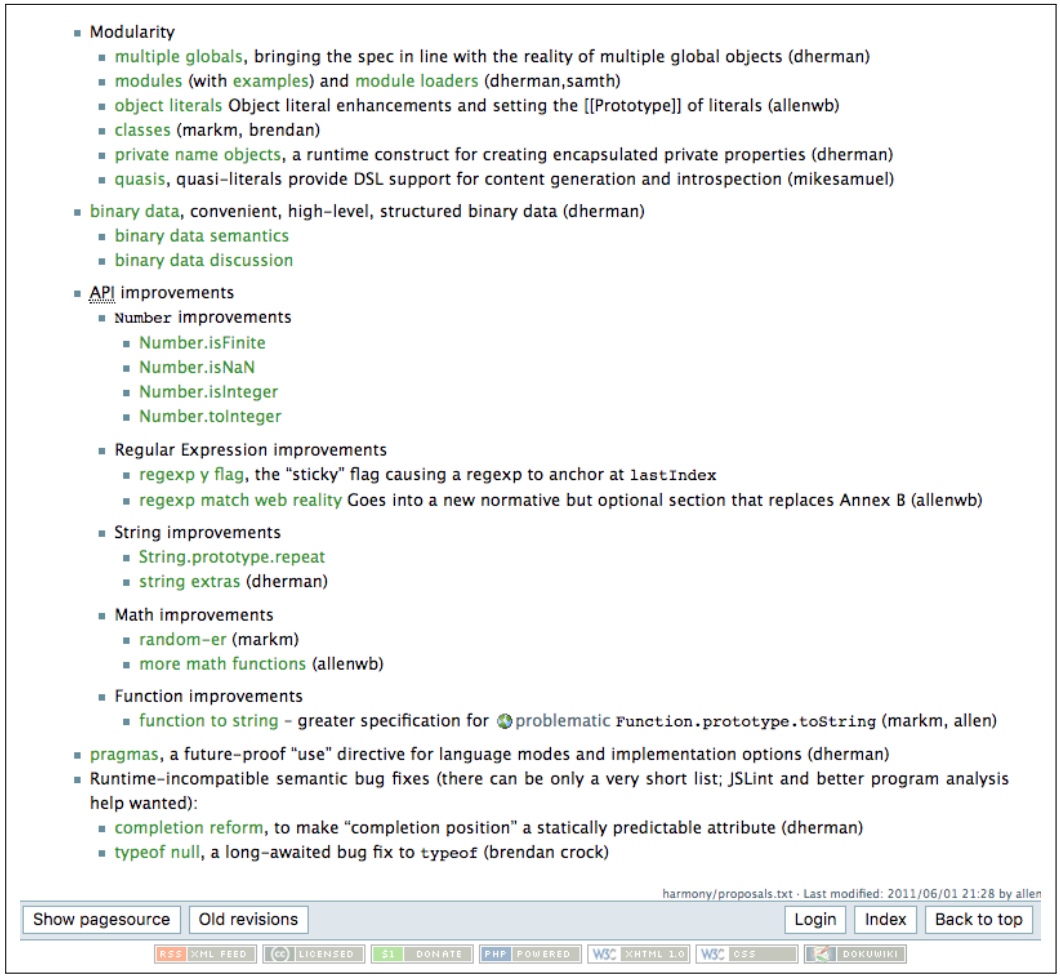
Lacking a formal language specification, the following list is not definitive, but it should reflect consensus achieved so far in TC39. Anything can be revisited for a good reason, of course.

The following list will grow, but not without bound before the next ECMA-262 Edition is constructed. Without prematurely triaging or using "ES6" (which might have to be used for a short-term edition, worst-case), for now let's focus on near-term proposals while keeping important longer-term strawman proposals warm.

— *Brendan Eich 2009/07/29 23:51*

Proposals

- Scoping, Binding, and Calling
  - Block scoped bindings (markm):
    - let, the new `var` but block-scoped and with better use-before-set semantics (dherman,markm)
    - const, for constant `let`-like bindings (dherman,markm)
    - block functions, `let`-scoped functions declared directly in block statements (markm)
  - destructuring assignment and binding declaration forms, based on object and array initialiser syntax (allen)
  - parameter default values, supplying default arguments to trailing optional parameters (allen)
  - rest parameters, for a trailing formal parameter capturing variable actual arguments as an array (allen)
  - spread, the `...` prefix operator for expanding an array actual parameter into its elements (allen)
  - proper tail calls (dherman)
- proxies, a "catchall" mechanism for intercepting property accesses generically (tomvc)
  - proxy defaultHandler, a default Proxy forwarding handler. (tomvc, markm)
  - proxies semantics, explains the semantics in terms of the ES5 spec's internal operations (tomvc)
  - extended object api standardizing some "missing" ES5 methods on Object. (tomvc)
- Collections
  - simple maps and sets (markm)
  - weak maps non-enumerable object-identity-based tables. Fixes a crucial memory leak in conventional weak-key tables. (markm)
  - egal (markm) an identity-testing operator inspired by Henry Baker's egal. (Not itself a collection, but some collections will rely on it.)
- Generation and Iteration
  - iterators, for better collections, more usable `for-in` constructs, and proxy enumeration that scales (brendan, dherman)
  - generators, Pythonic/Iconic generator functions that can yield multiple values while suspending their activation in between yields
  - generator expressions, convenient, expressive syntax for creating lazily-computed generators (brendan, dherman)
  - array comprehensions, convenient, expressive syntax for creating eagerly-computed arrays (brendan, dherman)



- Modularity
  - multiple globals, bringing the spec in line with the reality of multiple global objects (dherman)
  - modules (with examples) and module loaders (dherman,samth)
  - object literals Object literal enhancements and setting the [[Prototype]] of literals (allenwb)
  - classes (markm, brendan)
  - private name objects, a runtime construct for creating encapsulated private properties (dherman)
  - quasis, quasi-literals provide DSL support for content generation and introspection (mikesamuel)
- binary data, convenient, high-level, structured binary data (dherman)
  - binary data semantics
  - binary data discussion
- API improvements
  - `Number` improvements
    - Number.isFinite
    - Number.isNaN
    - Number.isInteger
    - Number.toInteger
  - Regular Expression improvements
    - regexp y flag, the "sticky" flag causing a regexp to anchor at `lastIndex`
    - regexp match web reality Goes into a new normative but optional section that replaces Annex B (allenwb)
  - String improvements
    - String.prototype.repeat
    - string extras (dherman)
  - Math improvements
    - random-er (markm)
    - more math functions (allenwb)
  - Function improvements
    - function to string - greater specification for problematic `Function.prototype.toString` (markm, allen)
- pragmas, a future-proof "use" directive for language modes and implementation options (dherman)
- Runtime-incompatible semantic bug fixes (there can be only a very short list; JSLint and better program analysis help wanted):
  - completion reform, to make "completion position" a statically predictable attribute (dherman)
  - typeof null, a long-awaited bug fix to `typeof` (brendan crock)

harmony/proposals.txt · Last modified: 2011/06/01 21:28 by allen

Show pagesource | Old revisions | Login | Index | Back to top

RSS XML FEED | LICENSED | $1 DONATE | PHP POWERED | W3C XHTML 1.0 | W3C CSS | DOKUWIKI

P TC39 POST ES6 PROCESS DEFINITION

[Weinstein and Wirfs-Brock 2013]

TC-39 Process

The ECMA TC-39 committee is responsible for evolving the ECMAScript programming language and authoring the specification. The committee operates by consensus and has discretion to alter the specification as it sees fit. However, the general process for making changes to the specification is as follows.

Development

Changes to the language are developed by way of a process which provides guidelines for evolving an addition from an idea to a fully specified feature, complete with acceptance tests and multiple implementations. There are four "maturity" stages. The TC-39 committee must approve acceptance for each stage.

Maturity Stages

	Stage	Purpose	Criteria	Acceptance signifies	Spec quality	Post-acceptance changes expected	Implementation types expected
0	***Strawman***	• *Allow input into the specification.*	• *None*	*N/A*	*N/A*	*N/A*	*N/A*
1	**Proposal**	• Make the case for the addition • Describe the shape of a solution • Identify potential challenges	• Identified "champion" who will advance the addition • Prose outlining the problem or need and the general shape of a solution. • Illustrative examples of usage • High-level API • Discussion of key algorithms, abstractions and semantics • Identification of potential "cross-cutting" concerns and implementation challenges/complexity.	The committee expects to devote time to examining the problem space, solutions and cross-cutting concerns	None	Major	Polyfills / demos
2	**Draft**	• Precisely describe the syntax and semantics using formal spec language	• Above • Initial spec text	The committee expects the feature to be developed and eventually included in the standard	Draft: all *major* semantics, syntax and API are covered, but TODOs, placeholders and editorial issues are expected	Incremental	Experimental
3	**Candidate**	• Indicate that further refinement will require feedback from implementations	• Above • Complete spec text • Designated reviewers have signed off on the current spec text. • The ECMAScript editor has signed off on the current spec text.	The solution is complete and no further work is possible without implementation experience, significant usage and external feedback.	Complete: all semantics, syntax and API are completed described	Limited: only those deemed critical based on implementation experience	Spec compliant
4	**Finished**	• Indicate that the addition is ready for inclusion in the formal ECMAScript standard	• Above • Test 262 acceptance tests have been written for mainline usage scenarios. • Two compatible implementations which pass the acceptance tests. • The ECMAScript editor has signed off on the current spec text.	The addition will be included in the soonest practical standard revision	Final: All changes as a result of implementation experience are integrated.	None	Shipping

Input into the process

Ideas for evolving the ECMAScript language are accepted in any form. Any discussion, idea or proposal for a change or addition which has not been submitted as a formal proposal is considered to be a "strawman" (stage 0) and has no acceptance requirements. Such submissions must either come from members of TC-39 or from non-members who have registered via ECMA International.

Spec revisions and scheduling

TC-39 *may* deliver to ECMA international a new revision of the ECMAScript language in March and September of every year. Additions which have been accepted by the committee as "finished" (stage 4) *may* be included in a new revision.

Status of in-process additions

TC-39 will maintain a list of in-process additions, along with the current maturity stage of each, on its website.

Spec Text

At stages "draft" (stage 2) and later, the semantics, API and syntax of an addition must be described as edits to the latest published ECMAScript standard, using the same language and conventions. The quality of the spec text expected at each stage is described

above.

Calls for implementation and feedback
When an addition is accepted at the "candidate" (stage 3) maturity level, the committee is signifying that it believes design work is complete and further refinement will require implementation experience, significant usage and external feedback.

Reviewers
Anyone can be a reviewer and submit feedback on an in-process addition. The committee may identify designated reviewers for acceptance at the "candidate" maturity stage. Designated reviewers should not be authors of the spec text for the addition and should have expertise applicable to the subject matter.

Eliding the process
The committee may elide the process based on the scope of a change under consideration as it sees fit.

Role of the editor
In-process additions will likely have spec text which is authored by a champion or a committee member other than the editor although in some case the editor may also be a champion with responsibility for specific features. The editor is responsible for the overall structure and coherence of the ECMAScript specification. It is also the role of the editor to provide guidance and feedback to spec text authors so that as an addition matures, the quality and completeness of its specification improves. It is also the role of the editor to integrate additions which have been accepted as "finished" (stage 4) into the a new revision of the specification.

Q THE EVOLUTION OF ECMASCRIPT PSEUDOCODE

Starting with ES1, ECMA-262 was specified using pseudocode algorithms consisting of numbered steps. For example, here is how the semantics of the conditional operator was specified in ES1:

The production *ConditionalExpression* : *LogicalORExpresion* **?** *AssignmentExpression* **:** *AssignmentExpresion* is evaluated as follows:

1. Evaluate *LogicalORExpression*.
2. Call GetValue(Result(1)).
3. ToBoolean(Result(2)).
4. If Result(3) is false, go to step 8.
5. Evaluate the first *AssignmentExpression*.
6. Call GetValue(Result(5)).
7. Return Result(6).
8. Evaluate the second *AssignmentExpression*.
9. Call GetValueResult(8)).
10. Return Result(9).

ES1–ES3 Pseudocode

The form is an introductory header followed by numbered steps. It can be characterized as machine-language–style pseudocode. It had all the same usability problems as coding in machine code. The goto control style and use of numeric labels for intermediate results made it difficult to understand and hard to maintain. This had not been a problem in ES1 where most of the algorithms were shorter than this example. It became a problem in ES3 which added several complex library functions that required longer algorithms, some with complex control flow.

Both attempts at ES4 abandoned the ES1–ES3 pseudocode for new formalisms. Waldemar Horwat for ES4$_1$ invented a more elaborate specification language. His version [Horwat 2003a] of the *ConditionalExpression* semantics was:

```
proc Eval[ConditionalExpression^β] (env: Environment, phase: Phase): ObjOrRef
  [ConditionalExpression^β ⇒ LogicalOrExpression^β ? AssignmentExpression^β_1 : AssignmentExpression^β_2] do
    a: Object ← readReference(Eval[LogicalOrExpression^β](env, phase), phase);
    if objectToBoolean(a) then
      return readReference(Eval[AssignmentExpression^β_1](env, phase), phase)
    else return readReference(Eval[AssignmentExpression^β_1](env, phase), phase)
    end if
end proc;
```

ES4$_1$ Pseudocode

$ES4_2$ did not progress to the point of specifying the *ConditionalExpression* production. However, *ConditionalExpression* was used by David Herman and Cormac Flanagan [2007] when they explained the intended ECMAScript specification usage of SML, using this example:

```
fun evalCondExpr (regs:REGS)
                 (cond:EXPR)
                 (thn:EXPR)
                 (els:EXPR)
  : VAL =
  let
      val v = evalExpr regs cond
      val b = toBoolean v
  in
      if b
      then evalExpr regs thn
      else evalExpr regs els
  end
```

$ES4_2$ ML Code

The ES5 working group wanted its specification to build upon and incrementally improve the notation and conventions used in ES1–ES3. However, for ES5 they needed to specify several additional complicated library functions. For better interoperability, they needed to write algorithms, some quite complex, for sections of the specification that had previously been inadequately described using incomplete or imprecise prose. Over several drafts they developed what Allen Wirfs-Brock called "structured programming" style pseudocode. The first major change was to introduce a "let statement" that allowed the naming of intermediate computational results. This convention had actually been used in two ES1 algorithms and in the ES3 regular expression matching algorithms but was not formally defined as part of the pseudocode conventions or applied pervasively throughout the specifications. Mark Miller introduced the second major change, which was to use structured control-flow statements and an outline indentation style to indicate nested control flows. In ES5, the *ConditionalExpression* semantics were expressed as:

The production *ConditionalExpression* : *LogicalORExpresion* **?** *AssignmentExpression* **:** *AssignmentExpresion* is evaluated as follows:

1. Let *lref* be the result of evaluating *LogicalORExpression*.
2. If ToBoolean(GetValue(*lref*)) is **true**, then
    a. Let *trueRef* be the result of evaluating the first *AssignmentExpression*.
    b. Return GetValue(*trueRef*).
3. Else
    a. Let *falseRef* be the result of evaluating the second *AssignmentExpression*.
    b. Return GetValue(*falseRef*).

ES5 Pseudocode

The step labels exist only to make it easy to reference individual steps in commentary. The algorithm conventions clause of the specification was revised to require use of the

new style. All new algorithms were written using that style and over the course of the project the ES editors rewrote all preëxisting algorithms to follow the new conventions.

The ES5 pseudocode conventions were used as the foundation for subsequent editions. ES6 used a simplified introductory header and made the semantics of exception propagation more explicit. The ES6 version of the *ConditionalExpression* semantics adds a single line to the algorithm, step 2:

Runtime Semantics: Evaluation

ConditionalExpression : *LogicalORExpresion* ? *AssignmentExpression* : *AssignmentExpresion*

1. Let *lref* be the result of evaluating *LogicalORExpression*.
2. ReturnIfAbrupt(*lref*)
3. If ToBoolean(GetValue(*lref*)) is **true**, then
   a. Let *trueRef* be the result of evaluating the first *AssignmentExpression*.
   b. Return GetValue(*trueRef*).
4. Else
   a. Let *falseRef* be the result of evaluating the second *AssignmentExpression*.
   b. Return GetValue(*falseRef*).

ES2015/ES6 Pseudocode

A goal of the ECMAScript specification is to be precise enough that a ECMAScript program cannot observe any differences in their behavior when run on different conforming implementations. One possible observable difference is the sequence in which an exception is propagated relative to other semantic actions that might have observable side effects.

Starting with ES3, the propagation of exceptions was modeled in the specification using a Completion Record abstraction. Completion Records were used in most specification algorithms and represent either a Normal Completion with a value or an Abrupt Completion such as a thrown exception. Prior to ES2015, the generation and discrimination of most Completion Records were implicit in the pseudocode. Returning a value from an algorithm implicitly generated a Normal Completion Record containing that value, and when the pseudocode that called such an algorithm accessed the return value it implicitly unwraps the value in a Normal Completion Record. But the specification did not explicitly define what happens when an Abrupt Completion is returned to a calling algorithm. Is an Abrupt Completion immediately propagated and the calling algorithm terminated or is the calling algorithm allowed to continue until it actually needs to use the returned value? Different implementations of ECMAScript made different choices. However, if an algorithm continues and performs operations with visible side effects this difference is observable by JavaScript programs. In the ES2015 specification this was clarified using additional steps such as step 2 in the ES2015 version of *ConditionalExpression*. ReturnIfAbrupt in step 2 is explicitly testing whether the evaluation of *LogicalORExpression* in step 1 produced an Abrupt Completion Record and if so it immediately returns that Abrupt Completion as the Completion Record for *ConditionalExpression*. If the result was a Normal Completion its value is unwrapped and assigned to *lref*. A ReturnIfAbrupt test is not needed for steps 3.a and 4.a because when GetValue is passed an Abrupt Completion as its argument it simply returns the Abrupt Completion as its own Completion Record.

ES2016 eliminates that extra line by consistently using a prefix question mark as a pseudocode operator that has semantics similar to ReturnIfAbrupt. It either propagates an Abrupt Completions or returns the unwrapped value of a Normal Completion:

Runtime Semantics: Evaluation

ConditionalExpression : *LogicalORExpresion* ? *AssignmentExpression* : *AssignmentExpresion*

1. Let *lref* be the result of evaluating *LogicalORExpression*.
2. If ToBoolean(?GetValue(*lref*)) is **true**, then
   a. Let *trueRef* be the result of evaluating the first *AssignmentExpression*.
   b. Return ?GetValue(*trueRef*).
3. Else
   a. Let *falseRef* be the result of evaluating the second *AssignmentExpression*.
   b. Return ?GetValue(*falseRef*).

ES2016 Pseudocode

REFERENCES

Adobe. 2007. *ActionScript Virtual Machine 2 (AVM2) Overview.* Adobe Systems Incorporated., 45 Park Avenue San Jose, CA 95110 (May). NON-ARCHIVAL http://wwwimages.adobe.com/content/dam/Adobe/en/devnet/actionscript/articles/avm2overview.pdf (BROKEN; also at Internet Archive 11 Aug. 2014 23:15:28).

Adobe. 2013. Flash Player penetration. Web page. NON-ARCHIVAL http://solutionpartners.adobe.com/products/player_census/flashplayer/ (BROKEN; also at Internet Archive 15 March 2013 11:45:33). Original date of the Web page is unknown. The data is from 2011.

ADsafe. 2007. Making JavaScript Safe for Advertising. Web site. NON-ARCHIVAL http://adsafe.org/ (also at Internet Archive 5 Jan. 2009 15:53:30, this is the earliest archived version of this site. Subsequent archived versions include additional information).

A.V. Aho, B.W. Kernighan, and P.J. Weinberger. 1988. *The AWK Programming Language.* Addison-Wesley Publishing Company. 9780201079814 lc87017566 Archived at https://archive.org/download/pdfy-MgN0H1joIoDVoIC7/The_AWK_Programming_Language.pdf

Jeremy Allaire. 2002. Macromedia Flash MX—A next-generation rich client. Macromedia white paper. March 2002. NON-ARCHIVAL https://download.macromedia.com/pub/flash/whitepapers/richclient.pdf (also at Internet Archive 2 Sept. 2018 12:14:22). This white paper contains the earliest known use of the term "Rich Internet Application".

Ben Alman. 2010. Immediately-Invoked Function Expression (IIFE). Blog post. 15 Nov. 2010. NON-ARCHIVAL http://benalman.com/news/2010/11/immediately-invoked-function-expression/ (also at Internet Archive 18 Nov. 2010 03:54:34).

Tim Anderson. 2007. Mark Anders Remembers Blackbird, and Other Microsoft Hits and Misses. Blog post on Tim Anderson's ITWriting Blog. 15 Oct. 2007. NON-ARCHIVAL http://www.itwriting.com/blog/363-mark-anders-remembers-blackbird-and-other-microsoft-hits-and-misses.html (also at Internet Archive 4 Oct. 2008 23:31:15).

ANSI X3. 1989. *American National Standard for Information Systems–programming language–C: ANSI X3.159–1989.* American National Standards Institute, NewYork, NewYork. Also ISO/IEC 9899:1990.

ANSI X3J20. 1998. *American National Standard for Information Technology–Programming Languages–Smalltalk: ANSI INCITS 319-1998.* American National Standards Institute, NewYork, NewYork.

Apple Computer. 1988. *Hypercard Script Language Guide: The Hypertalk Language.* Addison Wesley Publishing Company.

Erik Arvidsson. 2015. ECMAScript Object.observe spec. GitHub project repository. 14 Sept. 2015. NON-ARCHIVAL http://arv.github.io/ecmascript-object-observe/ (also at Internet Archive 20 Nov. 2015 01:02:05).

Jeremy Ashkenas. 2009. CoffeeScript, a little language that compiles to JavaScript. Happy Holidays, HN. Posting to Hacker News discussion forum. 24 Dec. 2009. NON-ARCHIVAL http://news.ycombinator.com/item?id=1014080 (also at Internet Archive 27 Dec. 2009 02:36:51).

Jeremy Ashkenas. 2010. CoffeeScript 1.0. Online manual. 24 Dec. 2010. NON-ARCHIVAL http://jashkenas.github.com/coffee-script/ (BROKEN; also at Internet Archive 30 Dec. 2010 11:17:19).

Jeremy Ashkenas. 2011. jashkenas / minimalist-classes.js. GitHub Gist. 31 Oct. 2011. NON-ARCHIVAL https://gist.github.com/jashkenas/1329619 (also at Internet Archive 13 Dec. 2013 04:17:13).

Jeremy Ashkenas et al. 2011. List of languages that compile to JS. Github Wiki Page. 6 Jan. 2011. NON-ARCHIVAL https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS (SUPERSEDED). Archived

at https://web.archive.org/web/20190327012411/https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS/eaca17540d9b66289dcc5f9bb8b6368f868c1e0a (this is the first version of the list).

Jeremy Ashkenas et al. 2018. List of languages that compile to JS. Github Wiki Page. 10 July 2018. NON-ARCHIVAL https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS/b14e7ef4c9963d50d39ed5443b5baa6ed0f63645 (link to current list; also at Internet Archive 27 March 2019 01:32:27, list as of July 2018).

Ihab A.B. Awad. 2010a. EMaker style modules for ECMAScript. Ecma/TC39/2010/004. 28 Feb. 2010. https://www.ecma-international.org/archive/ecmascript/2010/TC39/tc39-2010-004.pdf Presentation at TC39 meeting.

Ihab A.B. Awad. 2010b. Module proposals status. es-discuss mailing list. 18 May 2010. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2010-May/011220.html (also at Internet Archive 5 June 2014 01:09:25).

Ihab A.B. Awad. 2010c. Strawman: Modules Emaker Style. ecmascript.org wiki. 27 Jan. 2010. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:modules_emaker_style (BROKEN; also at Internet Archive 6 Feb. 2010 04:42:45).

Ihab A.B. Awad and Kris Kowal. 2009. Presentation on Modules. Ecma/TC39/2009/012. 29 Jan. 2009. https://www.ecma-international.org/archive/ecmascript/2009/TC39/tc39-2009-012.pdf

Babel Project. 2015. babeljs.io. Web site. 15 Feb. 2015. NON-ARCHIVAL https://babeljs.io/ (current site; also at Internet Archive 15 Feb. 2015 18:16:40, original site contents).

Adam Barth, Joel Weinberger, and Dawn Song. 2009. Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense. In *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, Fabian Monrose (Ed.). USENIX Association, 187–198. NON-ARCHIVAL https://www.usenix.org/legacy/events/sec09/tech/full_papers/barth.pdf (also at Internet Archive 14 Aug. 2017 10:41:16).

Jon Bentley. 1986. Programming Pearls: Little Languages. *Commun. ACM* 29, 8 (Aug.), 711–721. 0001-0782 https://doi.org/10.1145/6424.315691

Tim Berners-Lee. 2003. A Brief History of the Web. W3C web page. NON-ARCHIVAL https://www.w3.org/DesignIssues/TimBook-old/History.html (also at Internet Archive 15 Nov. 2019 11:53:52).

Nino Bilic. 2007. Happy 10th birthday, Outlook Web Access! Microsoft Exchange Team Blog. 13 June 2007. NON-ARCHIVAL https://techcommunity.microsoft.com/t5/Exchange-Team-Blog/Happy-10th-birthday-Outlook-Web-Access/ba-p/593150 (also at Internet Archive 8 July 2019 21:43:19).

John Borland. 2003. Browser wars: High price, huge rewards. *ZDNet.com* (15 April). NON-ARCHIVAL http://www.zdnet.com/article/browser-wars-high-price-huge-rewards/ (also at Internet Archive 21 Dec. 2014 07:45:10).

Borland International. 1996. Proposed JavaScript Extensions. Ecma/TC39/1996/006. 22 Nov. 1996. https://www.ecma-international.org/archive/ecmascript/1996/TC39/96-006.pdf

Bert Bos. 2005. "JavaScript, the worst invention ever". Web page. 8 May 2005. NON-ARCHIVAL http://www.phonk.net/Gedachten/JavaScript (also at Internet Archive 30 April 2006 04:12:47).

Jon Byous. 1998. Happy 3rd Birthday! java.sun.com website. 23 May 1998. NON-ARCHIVAL http://java.sun.com:80/features/1998/05/birthday.html (BROKEN; also at Internet Archive 24 Feb. 1999 05:34:07). This article was later renamed to Java Technology: An Early History.

Caja Project. 2012. Google Caja. developers.google.com website. 28 Feb. 2012. NON-ARCHIVAL https://developers.google.com/caja/ (also at Internet Archive 15 Nov. 2012 02:37:36).

Jonathan Cardy. 2011. A Collection of JavaScript Gotchas. The Code Project website. 2 Dec. 2011. NON-ARCHIVAL http://www.codeproject.com/Articles/182416/A-Collection-of-JavaScript-Gotchas (also at Internet Archive 3 Feb. 2012 00:35:12).

Patrick J. Caudill and Allen Wirfs-Brock. 1986. A Third Generation Smalltalk-80 Implementation. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications* (Portland, Oregon, USA) (*OOPSLA '86*). ACM, New York, NY, USA, 119–130. 0-89791-204-7 https://doi.org/10.1145/28697.28709

Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. 2003. The Maude 2.0 System. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications* (Valencia, Spain) (*RTA'03*). Springer-Verlag, Berlin, Heidelberg, 76–87. 3-540-40254-3 http://dl.acm.org/citation.cfm?id=1759148.1759156

Andrew Clinick. 1997. Proposal for Conditional Compile Support in ECMAScript. Ecma/TC39/1997/033. 11 July 1997. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-033.pdf

Andrew Clinick. 1999. ECMA TC39 and Working Group meetings – 29/30 March 1999. Ecma TC39wg tcn9903. 30 March 1999. https://www.ecma-international.org/archive/ecmascript/1999/TC39WG/tcn9903.htm

Andrew Clinick. 2000. Introducing JScript .NET. Micrsoft Scripting Clinic column. 14 July 2000. NON-ARCHIVAL https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/scripting-articles/ms974588(v=msdn.10) (also at Internet Archive 30 Aug. 2018 11:06:29).

CommonJS Project. 2009. Modules/1.0. CommonJS Wiki. 22 April 2009. NON-ARCHIVAL http://wiki.commonjs.org/index.php?title=Modules/1.0&oldid=489 (also at Internet Archive 7 April 2019 19:26:43).

William R. Cook. 2007. AppleScript. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, California) (*HOPL III*). ACM, New York, NY, USA, 1–1–1–21. 978-1-59593-766-7 https://doi.org/10.1145/1238844.1238845

Mike Cowlishaw (Ed.). 1998. *ECMA-262, 2nd Edition: ECMAScript Language Specification.* Ecma International, Geneva, Switzerland (Aug.). https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%202nd%20edition,%20August%201998.pdf

Mike Cowlishaw (Ed.). 1999a. *ECMA-262, 3rd Edition: ECMAScript Language Specification.* Ecma International, Geneva, Switzerland (Dec.). https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf

Mike Cowlishaw. 1999b. ECMAScript E3 draft status [1999.04.08]. Ecma/TC39/1999/003. 8 April 1999. https://www.ecma-international.org/archive/ecmascript/1999/TC39/9t39-003.pdf

Mike Cowlishaw. 1999c. ECMAScript E3 draft status [1999.08.03]. Ecma TC39wg 990803-e3status. 3 Aug. 1999. https://www.ecma-international.org/archive/ecmascript/1999/TC39WG/990803-e3status.htm

Mike Cowlishaw et al. 1998. ECMAScript Language Specification, Version 2 Working Draft (with change bars). Ecma TC39 working document. 22 April 1998. https://www.ecma-international.org/archive/ecmascript/1998/TC39WG/es2D9804.pdf

Douglas Crockford. 2001a. JavaScript: The World's Most Misunderstood Programming Language. Web page. NON-ARCHIVAL http://www.crockford.com/javascript/javascript.html (also at Internet Archive 10 Aug. 2002 19:25:30).

Douglas Crockford. 2001b. JSLINT The JavaScript Verifier. Web page. NON-ARCHIVAL http://www.crockford.com:80/javascript/lint.html (SUPERSEDED; also at Internet Archive 1 May 2002 09:03:02). page relocated in 2005: NON-ARCHIVAL http://www.jslint.com/lint.html

Douglas Crockford. 2001c. JSMIN The JavaScript Minifier. Web page. 11 Sept. 2001. NON-ARCHIVAL http://www.crockford.com/javascript/jsmin.html (current version; also at Internet Archive 5 April 2002 14:04:16, original version).

Douglas Crockford. 2001d. Private Members in JavaScript. Web page. NON-ARCHIVAL http://www.crockford.com/javascript/private.html (also at Internet Archive 10 Aug. 2002 19:33:24).

Douglas Crockford. 2001e. A Survey of the JavaScript Programming Language. Web page. NON-ARCHIVAL http:/www.crockford.com:80/javascript/survey.html (also at Internet Archive 5 June 2002 05:31:45).

Douglas Crockford. 2002a. Classical Inheritance in JavaScript. Web page. Sept. 2002. NON-ARCHIVAL http://www.crockford.com:80/javascript/inheritance.html (also at Internet Archive 16 Oct. 2002 17:12:01).

Douglas Crockford. 2002b. Introducing JSON. Web page. Dec. 2002. NON-ARCHIVAL http://www.crockford.com/JSON/index.html (BROKEN; also at Internet Archive 28 Feb. 2003 03:41:47).

Douglas Crockford. 2002c. JSON in JavaScript. Web page. Dec. 2002. NON-ARCHIVAL http://www.crockford.com:80/JSON/js.html (BROKEN). Archived at hhttps://web.archive.org/web/20030111194120/http://www.crockford.com:80/JSON/js.html

Douglas Crockford. 2002d. Recommendations for Modifications to the ECMAScript Language Specification. Web page. 4 Jan. 2002. NON-ARCHIVAL http://www.crockford.com:80/javascript/recommend.html (SUPERSEDED; also at Internet Archive 5 April 2002 14:32:50).

Douglas Crockford. 2003. The Little JavaScripter. Web page. NON-ARCHIVAL http://www.crockford.com/javascript/little.html (SUPERSEDED; also at Internet Archive 8 April 2003 00:03:36).

Douglas Crockford. 2006. Prototypal Inheritance in JavaScript. Web page. 7 June 2006. NON-ARCHIVAL http://www.crockford.com/javascript/prototypal.html (updated 7 March 2008; also at Internet Archive 20 Oct. 2008 05:59:23). Original version archived on 16 June 2006 https://web.archive.org/web/20060616111939/http://javascript.crockford.com/prototypal.html

Douglas Crockford. 2007a. Re: ECMAScript 4 Language Overview White Paper (21 Oct. 2007, 11:31 AM). Message to TC39-TG1 private mailing list. Archived by Ecma International.

Douglas Crockford. 2007b. Recommendations for Modifications to the ECMAScript Language Specification (2007-4-21). Web page. 21 April 2007. NON-ARCHIVAL http://www.crockford.com:80/javascript/recommend.html (SUPERSEDED; also at Internet Archive 7 June 2007 17:26:21).

Douglas Crockford. 2007c. Recommendations for Modifications to the ECMAScript Language Specification (2007-4-4). Web page. 4 April 2007. NON-ARCHIVAL http://www.crockford.com:80/javascript/recommend.html (SUPERSEDED; also at Internet Archive 6 April 2007 06:37:23).

Douglas Crockford. 2007d. Recommendations for Modifications to the ECMAScript Language Specification (2007-6-21). Web page. 21 June 2007. NON-ARCHIVAL http://www.crockford.com:80/javascript/recommend.html (also at Internet Archive 14 Aug. 2007 04:51:17).

Douglas Crockford. 2008a. es3.1 goals. ecmascript.org wiki. 26 Feb. 2008. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=es3.1:es3.1_goals (BROKEN; also at Internet Archive 13 Oct. 2016 19:48:29).

Douglas Crockford. 2008b. *JavaScript: The Good Parts.* O'Reilly Media, Inc. 0596517742

Douglas Crockford. 2008c. The Only Thing We Have To Fear Is Premature Standardization. Yahoo! User Interface Blog. 14 Aug. 2008. NON-ARCHIVAL http://yuiblog.com/blog/2008/08/14/premature-standardization/ (also at Internet Archive 17 Aug. 2008 15:15:45).

Douglas Crockford. 2008d. Secure ECMAScript. Ecma/TC39/2008/086. Aug. 2008. https://www.ecma-international.org/archive/ecmascript/2008/TC39/tc39-2008-086-sesintro.pdf

Douglas Crockford. 2019a. json.org. Website. May 2019. NON-ARCHIVAL https://json.org/ Archived at https://web.archive.org/web/20200216094715/https://www.json.org/json-en.html

Douglas Crockford. 2019b. Minify. Blog post. 5 May 2019. NON-ARCHIVAL https://www.crockford.com/minify.html (also at Internet Archive 11 May 2019 18:59:04).

Douglas Crockford, Pratap Lakshman, and Allen Wirfs-Brock. 2007. Proposal to Refocus TC39-TG1 On the Maintenance of the ECMAScript 3rd Edition Specification. ecmascript.org wiki. 16 March 2007. https://ecma-international.org/archive/ecmascript/2007/misc/proposal_to_refocus_tc39-tg1.pdf

Ryan Dahl. 2009. (Non-archival) Node.js. Video of presentation at the European JavaScript Conference. 7 Nov. 2009. NON-ARCHIVAL https://youtu.be/ztspvPYybIY (retrieved 7 March 2019)

Kevin Dangoor. 2009. What Server Side JavaScript needs. Blog post. 29 Jan. 2009. NON-ARCHIVAL http://www.blueskyonmars.com/2009/01/29/what-server-side-javascript-needs/ (also at Internet Archive 31 Jan. 2009 20:07:16).

Kevin Dangoor. 2010. CommonJS: the First Year. Blog post. 29 Jan. 2010. NON-ARCHIVAL http://www.blueskyonmars.com/2010/01/29/commonjs-the-first-year/ (also at Internet Archive 1 Feb. 2010 03:18:49).

Olivier Danvy. 2005. Transcript of Q&A following Brendan Eich ICFP 2005 keynote. Comment on blog post. 27 Oct. 2005. NON-ARCHIVAL http://brendaneich.com/2005/10/recap-and-prelude/#comment-336 (also at Internet Archive 5 March 2011 18:57:51).

Ilana Dashevsky and Vicki Balzano. 2008. James Webb Space Telescope ground to flight interface design. In *2008 IEEE Aerospace Conference*. IEEE, IEEE, 1–7.

Domenic Denicola. 2014. ModuleImport. es-discuss mailing list. 19 June 2014. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2014-June/037951.html (also at Internet Archive 23 July 2014 23:15:52).

Domenic Denicola. 2016. Adding JavaScript modules to the web platform. Blog post on The WHATWG Blog. 13 April 2016. NON-ARCHIVAL https://blog.whatwg.org/js-modules (also at Internet Archive 14 April 2016 00:50:24).

Ken Dickey. 1992. Scheming with objects. *AI Expert* 7, 10, 24–33. NON-ARCHIVAL http://ftp.cs.indiana.edu/pub/scheme-repository/doc/pubs/swob.txt (BROKEN; also at Internet Archive 16 Dec. 1999 14:28:10).

Darcy DiNucci. 1999. Fragmented Future. *Print* 53, 4. NON-ARCHIVAL http://darcyd.com/fragmented_future.pdf (also at Internet Archive 16 Nov. 2011 23:46:36).

Chris Dollin. 2002. Spice language manual. HP Labs Technical Report HPL-2002-229. 30 Oct. 2002. NON-ARCHIVAL http://www.hpl.hp.com/techreports/2002/HPL-2002-229.pdf (also at Internet Archive 31 Aug. 2003 04:58:04).

Jeff Dyer. 2008a. ES4 work. ES4-discuss mailing list. 15 Feb. 2008. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2008-February/005335.html (also at Internet Archive 5 June 2014 05:48:10).

Jeff Dyer. 2008b. Packages must go. ES4-discuss mailing list. 17 April 2008. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2008-April/006183.html (also at Internet Archive 5 June 2014 05:21:32).

Ecma International. 1996a. Invitation and venue for the start-up meeting on a project on Java Script. Ecma/TC39/1996/001. 19 Sept. 1996. https://www.ecma-international.org/archive/ecmascript/1996/TC39/96-001.pdf

Ecma International. 1996b. Minutes for the Ecma Co-ordinating Committee, Sept. 19-20, 1996. Ecma/GA/1996/083. 19 Sept. 1996. https://www.ecma-international.org/archive/ecmascript/1996/GA/96-083-excerpt.pdf

Ecma International. 1997. Minutes for the 73rd General Assembly, 26-27 June 1997. Ecma/GA/1997/063 (excerpts). 26 June 1997. https://www.ecma-international.org/archive/ecmascript/1997/GA/97-063-excerpt.pdf

Ecma International. 1999. Minutes for the 78th General Assembly, 16-17 Dec. 1997. Ecma/GA/1999/137 (excerpts). 16 Dec. 1999. https://www.ecma-international.org/archive/ecmascript/1999/GA/9ga-137-excerpts.pdf

Ecma International. 2004. Ecma International Approves ECMAScript for XML. Press Release Ecma/GA/2004/148. 20 July 2004. https://www.ecma-international.org/archive/ecmascript/2004/GA/ga-2004-148.pdf

Ecma International. 2007a. Minutes for the meeting of the Co-ordinating Committee, 23-24 October 2007. Ecma/GA/2007/202 (excerpts). 24 Oct. 2007. https://www.ecma-international.org/archive/ecmascript/2007/GA/ga-2007-202-excerpt.pdf

Ecma International. 2007b. Minutes for the meeting of the Co-ordinating Committee, 9-10 May 2007. Ecma/GA/2007/088 (excerpts). 10 May 2007. https://www.ecma-international.org/archive/ecmascript/2007/GA/ga-2007-088-excerpt.pdf

Ecma International. 2008. Press Release: TC39 coalesces on future direction of Web Programming Language. Ecma/TC39/2008/073. 19 Aug. 2008. https://www.ecma-international.org/archive/ecmascript/2008/TC39/tc39-2008-073.pdf

Ecma International. 2009a. Ecma International approves major revision of ECMAScript. Press Release Ecma/GA/2009/204. 15 Dec. 2009. https://ecma-international.org/news/PressReleases/PR_Ecma%20approves%20major%20revision%20of%20ECMAScript.htm Archived at https://www.ecma-international.org/archive/ecmascript/2009/GA/ga-2009-204.pdf

Ecma International. 2009b. Minutes of the 98th meerting of the Ecma General Assembly, 3 December 2009. Ecma/GA/2009/203 (excerpts). 3 Dec. 2009. https://www.ecma-international.org/archive/ecmascript/2009/GA/ga-2009-203-excerpt.pdf

Ecma International. 2015a. At the June 17, 2015 Ecma General Assembly in Montreux, ECMA-262 6th edition - ECMAScript 2015 Language Specification and ECMA-402 2nd edition - ECMAScript 2015 Internationalization API have been adopted. Press Release. 2 July 2015. https://www.ecma-international.org/news/Publication%20of%20ECMA-262%206th%20edition.htm

Ecma International. 2015b. Minutes of the 98th meerting of the Ecma General Assembly, 17 June 2015. Ecma/GA/2015/065-Rev1 (excerpts). June 2015. https://www.ecma-international.org/archive/ecmascript/2015/GA/ga-2015-068-Rev1-excerpt.pdf

Brendan Eich. 2004. The non-world non-wide non-web. Blog post. 4 June 2004. NON-ARCHIVAL http://brendaneich.com/2004/06/the-non-world-non-wide-non-web/ (also at Internet Archive 11 April 2011 02:14:44).

Brendan Eich. 2005a. JavaScript 1, 2, and in between. Blog post. 13 June 2005. NON-ARCHIVAL http://brendaneich.com/2005/06/javascript-1-2-and-in-between/ (also at Internet Archive 7 Aug. 2011 19:55:50).

Brendan Eich. 2005b. JavaScript at Ten Years. ICFP'05 Keynote presentation slide deck. 26 Sept. 2005. NON-ARCHIVAL http://www-archive.mozilla.org/js/language/ICFP-Keynote.ppt (also at Internet Archive 25 July 2011 10:53:50).

Brendan Eich. 2005c. JS2 Design Notes. Blog post. 9 Nov. 2005. NON-ARCHIVAL http://brendaneich.com/2005/11/js2-design-notes/ (also at Internet Archive 7 Aug. 2011 19:56:35).

Brendan Eich. 2005d. Recap and Prelude. Blog post. 27 Oct. 2005. NON-ARCHIVAL http://brendaneich.com/2005/10/recap-and-prelude/ (also at Internet Archive 5 March 2011 18:57:51).

Brendan Eich. 2006a. JavaScript2 And The Future Of The Web. XTech 2006 presentation slide deck. 19 May 2006. NON-ARCHIVAL http://developer.mozilla.org:80/presentations/xtech2006/javascript/ (BROKEN; also at Internet Archive 27 May 2006 21:59:09).

Brendan Eich. 2006b. Will there be a suggested file suffix for es4? es-discuss mailing list. 3 Oct. 2006. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2006-October/000133.html (also at Internet Archive 22 May 2011 17:00:37).

Brendan Eich. 2007a. New Projects. Brendan's Roadmap Updates blog post. 25 July 2007. NON-ARCHIVAL http://weblogs.mozillazine.org/roadmap/archives/2007/07/new_projects.html (BROKEN; also at Internet Archive 23 Aug. 2007 19:42:37).

Brendan Eich. 2007b. RE: Refocus (16 March 2007, 10:42 PM). Message to TC39-TG1 private mailing list. Archived by Ecma International.

Brendan Eich. 2007c. RE: Refocus (16 March 2007, 4:22 PM). Message to TC39-TG1 private mailing list. Archived by Ecma International.

Brendan Eich. 2007d. TG1 Convener's Report to TC39. Ecma/TC39-TG1/2007/001. 7 Sept. 2007. https://www.ecma-international.org/archive/ecmascript/2007/TG1/tc39-tg1-2007-001.pdf

Brendan Eich. 2008a. Allen's lambda syntax proposal. es-discuss mailing list. 29 Nov. 2008. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2008-November/008216.html (also at Internet Archive 18 July 2010 15:22:18).

Brendan Eich. 2008b. ECMAScript Harmony. es-discuss mailing list. 13 Aug. 2008. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2008-August/006837.html (also at Internet Archive 22 May 2011 17:16:41).

Brendan Eich. 2008c. Popularity. Blog post. April 2008. NON-ARCHIVAL http://brendaneich.com/2008/04/ (also at Internet Archive 4 Feb. 2015 16:07:55).

Brendan Eich. 2009a. harmony:harmony. ecmascript.org wiki. 27 July 2009. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=harmony:harmony (BROKEN; also at Internet Archive 18 Aug. 2009 15:34:47).

Brendan Eich. 2009b. Improving ECMAScript as a compilation target. es-discuss mailing list. 4 May 2009. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2009-May/009168.html (also at Internet Archive 5 June 2014 02:53:59).

Brendan Eich. 2009c. Presentation on modules. es-discuss mailing list. 7 Nov. 2009. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2009-November/010155.html (also at Internet Archive 5 June 2014 02:18:05).

Brendan Eich. 2009d. Strawman: catchalls. ecmascript.org wiki. 4 May 2009. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:catchalls (BROKEN; also at Internet Archive 29 Sept. 2009 03:34:07).

Brendan Eich. 2010a. harmony:harmony. ecmascript.org wiki. 28 April 2010. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=harmony:harmony (BROKEN; also at Internet Archive 1 July 2010 21:41:35).

Brendan Eich. 2010b. three small proposals: the bikeshed cometh! es-discuss mailing list. 29 April 2010. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2010-April/011010.html (also at Internet Archive 5 June 2014 01:24:40).

Brendan Eich. 2011a. BrendanEich/minimalist-classes.js Less minimalism, richer leather. GitHub Gist. 1 Nov. 2011. NON-ARCHIVAL https://gist.github.com/BrendanEich/1332193 (also at Internet Archive 24 July 2014 03:50:15).

Brendan Eich. 2011b. Harmony of My Dreams. Blog post. Jan. 2011. NON-ARCHIVAL http://brendaneich.com/2011/01/harmony-of-my-dreams/ (also at Internet Archive 30 Jan. 2011 23:34:27).

Brendan Eich. 2011c. My JSConf.US Presentation. Blog post. 4 May 2011. NON-ARCHIVAL http://brendaneich.com:80/2011/05/my-jsconf-us-presentation (also at Internet Archive 8 May 2011 05:20:26).

Brendan Eich. 2011d. New JavaScript Engine Module Owner. Blog post. 21 June 2011. NON-ARCHIVAL https://brendaneich.com/2011/06/ (also at Internet Archive 20 March 2019 11:24:31).

Brendan Eich. 2011e. New JavaScript Engine Module Owner. Slide deck for presentation at CapitolJS conference. 18 Sept. 2011. NON-ARCHIVAL https://www.slideshare.net/BrendanEich/jslol-9539395 (also at Internet Archive 9 Oct. 2011 04:23:38).

Brendan Eich. 2011f. Strawman: arrow function syntax. ecmascript.org wiki. 2 May 2011. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:arrow_function_syntax (BROKEN; also at Internet Archive 9 May 2011 07:01:04).

Brendan Eich. 2011g. Strawman: block lambda revival. ecmascript.org wiki. 20 May 2011. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:block_lambda_revival (BROKEN; also at Internet Archive 15 June 2011 16:18:17).

Brendan Eich. 2012a. Class declarations. es-discuss mailing list. 16 March 2012. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2012-March/021259.html (also at Internet Archive 4 June 2014 18:01:55).

Brendan Eich. 2012b. Harmony: arrow function syntax. ecmascript.org wiki. 26 May 2012. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=harmony:arrow_function_syntax (BROKEN; also at Internet Archive 7 June 2012 07:47:19).

Brendan Eich. 2012c. u. es-discuss mailing list. 15 March 2012. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2012-March/021232.html (also at Internet Archive 4 June 2014 17:58:22).

Brendan Eich. 2013. Value Objects. Ecma/TC39/2013/040. 25 July 2013. https://www.ecma-international.org/archive/ecmascript/2013/TC39/tc39-2013-040.pdf

Brendan Eich et al. 1998. SpiderMonkey JS 1.4 source code. Mozilla snapshot of SpiderMonkey JS 1.4 source code. Oct. 1998. NON-ARCHIVAL https://dxr.mozilla.org/classic/source/js/src (retrieved 5 June 2019; also at Internet Archive 5 June 2019 18:58:32).

Brendan Eich et al. 2008. TC39 coalesces on future direction of Web Programming Language. Ecma/TC39/2008/074. 19 Aug. 2008. https://www.ecma-international.org/archive/ecmascript/2008/TC39/tc39-2008-074.pdf

Brendan Eich et al. 2012. The Narcissus meta-circular JavaScript interpreter. GitHub repository. Feb. 2012. NON-ARCHIVAL https://github.com/mozilla/narcissus/ (also at Internet Archive 7 Aug. 2013 22:04:22).

Brendan Eich and C. Rand McKinney. 1996. JavaScript Language Specification. Ecma/TC39/1996/002. 18 Nov. 1996. https://www.ecma-international.org/archive/ecmascript/1996/TC39/96-002.pdf

ES5conform. 2009. ECMAScript 5 Conformance Suite. Codeplex project. 22 June 2009. NON-ARCHIVAL http://es5conform.codeplex.com (also at Internet Archive 1 July 2009 02:26:40). Zip file of final ES5conform project artifacts including source code https://web.archive.org/web/20180705194718/https://codeplexarchive.blob.core.windows.net/archive/projects/ES5conform/ES5conform.zip

Erik Fair. 1998. JavaScript Must Be Eradicated From The Web. Web page. 4 Dec. 1998. NON-ARCHIVAL http://www.clock.org/~fair/opinion/javascript-is-evil.html (also at Internet Archive 25 May 2000 09:07:14).

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-order Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming* (Pittsburgh, PA, USA) (*ICFP '02*). ACM, New York, NY, USA, 48–59. 1-58113-487-8 https://doi.org/10.1145/581478.581484

Cormac Flanagan. 2006. Hybrid Type Checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) (*POPL '06*). ACM, New York, NY, USA, 245–256. 1-59593-027-2 https://doi.org/10.1145/1111037.1111059

Cormac Flanagan. 2008. ES-Harmony Class System Proposal. ecmascript.org wiki. Nov. 2008. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=strawman:classes (BROKEN; also at Internet Archive 8 Sept. 2010 00:46:06).

Richard P Gabriel. 1990. Lisp: Good News, Bad News, How to Win Big (keynote), European Conference on the Practical Applications of Lisp, Cambridge University, Cambridge, England, March 1990. reprinted in AI Expert, June 1991, pp. 31–39. March 1990. NON-ARCHIVAL http://www.dreamsongs.com/WorseIsBetter.html (also at Internet Archive 2 July 2019 10:03:53).

Andreas Gal et al. 2009. Trace-based just-in-time type specialization for dynamic languages. *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation - PLDI '09*. 9781605583921 https://doi.org/10.1145/1542476.1542528

Jesse James Garrett. 2005. Ajax: A New Approach to Web Applications. Feb. 2005. NON-ARCHIVAL https://www.adaptivepath.org/ideas/ajax-new-approach-web-applications/ (BROKEN; also at Internet Archive 10 Sept. 2015 07:23:59).

Bill Gates. 1995. The Internet Tidal Wave. Internal Microsoft Memo published on Letters of Note web site. 26 May 1995. NON-ARCHIVAL http://www.lettersofnote.com/2011/07/internet-tidal-wave.html (also at Internet Archive 24 July 2011 18:44:30).

Jonathan Gay. 2006. History of Flash. Adobe Showcase. May 2006. NON-ARCHIVAL http://www.adobe.com/macromedia/events/john_gay/ (BROKEN; also at Internet Archive 3 May 2006 17:53:13).

General Magic. 1995. *Telescript Language Reference*. General Magic Inc., Sunnyvale, CA (Oct.). NON-ARCHIVAL http://bitsavers.org/pdf/generalMagic/Telescript_Language_Reference_Oct95.pdf (also at Internet Archive 5 May 2010 12:51:10).

Bill Gibbons et al. 1999. ECMAScript Language Specification, Edition 3 Final Draft. Ecma TC39 working document. 25 Aug. 1999. https://www.ecma-international.org/archive/ecmascript/1999/TC39WG/990825-e3-final.pdf

Richard Gillam. 1998. I18N meeting minutes. Ecma TC39 working document. 18 Nov. 1998. https://www.ecma-international.org/archive/ecmascript/1998/TC39WG/981118-i18n9811.htm

Richard Gillam et al. 1999a. Proposal for Improving Internationalization Support in ECMAScript 2.0 (Version 0.3). Ecma TC39 working document. 15 Jan. 1999. https://www.ecma-international.org/archive/ecmascript/1999/TC39WG/990115-i18n9901.pdf

Richard Gillam et al. 1999b. Proposal for Improving Internationalization Support in ECMAScript 2.0 (Version 1.0). Ecma TC39 working document. 29 April 1999. https://www.ecma-international.org/archive/ecmascript/1999/TC39WG/990429-i18n9904.pdf

Google. 2006. Google Web Toolkit(Beta). Google Code Project Page. May 2006. NON-ARCHIVAL http://code.google.com/webtoolkit/ (SUPERSEDED; also at Internet Archive 23 May 2006 14:13:21). NON-ARCHIVAL http://code.google.com/webtoolkit/ (current version).

Google. 2008a. V8 Benchmark Suite – version 1. Web page. Sept. 2008. NON-ARCHIVAL http://code.google.com/apis/v8/run.html (SUPERSEDED; also at Internet Archive 4 Sept. 2008 20:13:17).

Google. 2008b. V8 JavaScript Engine: Design Elements. Web page. 4 Sept. 2008. NON-ARCHIVAL http://code.google.com/apis/v8/design.html (SUPERSEDED; also at Internet Archive 4 Sept. 2008 20:17:14).

Google. 2012a. Chromium. Project website. Jan. 2012. NON-ARCHIVAL https://www.chromium.org/Home (also at Internet Archive 4 Jan. 2012 01:11:48).

Google. 2012b. Chromium with the Dart VM. Web page. Feb. 2012. NON-ARCHIVAL http://www.dartlang.org:80/dartium/index.html (SUPERSEDED; also at Internet Archive 18 Feb. 2012 22:45:29).

J. Gosling, B. Joy, and G.L. Steele. 1996. *The Java Language Specification.* Addison-Wesley. 9780201634518 96031170

Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. 2019. How to evaluate the performance of gradual type systems. *J. Funct. Program.* 29, e4. https://doi.org/10.1017/S0956796818000217

Sacha Greif and Raphaël Benitte. 2019. State of JavaScript Survey – JavaScript Flavors. Web page. NON-ARCHIVAL https://2019.stateofjs.com/javascript-flavors/ (also at Internet Archive 30 Jan. 2020 23:16:05). The archived pages do not show the data visualizations visible on the original website. The two relevant visualizations are separately archived ats https://web.archive.org/web/20200208164728/https://2019.stateofjs.com/images/captures/javascript_flavors_experience_ranking.png and https://web.archive.org/web/20200208155436/https://2019.stateofjs.com/images/captures/javascript_flavors_section_overview.png.

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017.* 9781450349888 https://doi.org/10.1145/3062341.3062363

Peter Hallam and Alex Russell. 2011. The Future of JS and You. Video of presentation at NodeConf 2011. 12 May 2011. NON-ARCHIVAL https://youtu.be/ntDZa7ekFEA (retrieved 6 March 2019)

Christian Plesner Hansen. 2009. Launching Sputnik into Orbit. Chromium Blog. 29 June 2009. NON-ARCHIVAL http://blog.chromium.org/2009/06/launching-sputnik-into-orbit.html (also at Internet Archive 1 July 2009 03:41:42).

Lars T Hansen. 2007a. Compatibility Between ES3 and Proposed ES4. Ecma/TC39-TG1/2007/046. 29 Nov. 2007. https://www.ecma-international.org/archive/ecmascript/2007/TG1/tc39-tg1-2007-046.pdf

Lars T Hansen. 2007b. ECMAScript 4 Language Overview White Paper (21 Oct. 2007, 10:32 AM). Message to TC39-TG1 private mailing list. Archived by Ecma International.

Lars T Hansen. 2007c. ECMAScript 4th Edition – Project Editor's Report. Ecma/TC39-TG1/2007/044. 8 Nov. 2007. https://www.ecma-international.org/archive/ecmascript/2007/TG1/tc39-tg1-2007-044.pdf

Lars T Hansen. 2007d. Evolutionary Programming and Gradual Typing in ECMAScript 4. Ecma/TC39-TG1/2007/045. 30 Nov. 2007. https://www.ecma-international.org/archive/ecmascript/2007/TG1/tc39-tg1-2007-045.pdf

Lars T Hansen. 2007e. Proposed ECMAScript 4th Edition – Language Overview (Revised 23 October 2007). Originally published at www.ecmpscript.org. 23 Oct. 2007. https://www.ecma-international.org/archive/ecmascript/2007/misc/overview.pdf

Lars T Hansen. 2008. Proposed ES4 draft 1. ES4-discuss mailing list (16 May 2008. 16 May 2008. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2008-May/006312.html (also at Internet Archive 5 June 2014 04:58:43).

Lars T Hansen et al. 2008a. Proposed ECMAScript 4th Edition Specification, Core Language Draft 1. Ecma/TC39/2008/042. 16 May 2008. https://www.ecma-international.org/archive/ecmascript/2008/TC39/tc39-2008-042.pdf

Lars T Hansen et al. 2008b. Proposed ECMAScript 4th Edition Specification Draft 1. Ecma/TC39/2008/040. 16 May 2008. https://www.ecma-international.org/archive/ecmascript/2008/TC39/tc39-2008-040.pdf

Lars T Hansen et al. 2008c. Proposed ECMAScript 4th Edition Specification, Surface Syntax Draft 1. Ecma/TC39/2008/041. 16 May 2008. https://www.ecma-international.org/archive/ecmascript/2008/TC39/tc39-2008-041.pdf

Lars T Hansen and Jeff Dyer. 2008. Features to Defer From Proposed ECMAScript 4. ES4-discuss mailing list. 26 Feb. 2008. NON-ARCHIVAL http://mail.mozilla.org/pipermail/es-discuss/attachments/20080226/53160c4c/attachment-0002.obj (also at Internet Archive 5 June 2014 05:52:40).

David Harel. 2007. Statecharts in the Making: A Personal Account. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, California) (*HOPL III*). ACM, New York, NY, USA, 5–1–5–43. 978-1-59593-766-7 https://doi.org/10.1145/1238844.1238849

David Herman. 2005. ClassicJavaScript CEKS semantics. Web page. July 2005. NON-ARCHIVAL http://www.ccs.neu.edu/home/dherman/javascript/ (also at Internet Archive 18 July 2007 12:44:22).

David Herman. 2007. ECMAScript Edition 4 Reference Implementation. Lambda the Ultimate weblog post. 8 June 2007. NON-ARCHIVAL http://lambda-the-ultimate.org/node/2289 (also at Internet Archive 11 June 2007 16:15:19).

David Herman. 2008. Strawman: lambdas. ecmascript.org wiki. 13 Oct. 2008. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:lambdas (BROKEN; also at Internet Archive 15 Oct. 2008 12:56:19).

David Herman. 2010a. Harmony: Generator Expressions. ecmascript.org wiki. 25 June 2010. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=harmony:generator_expressions (BROKEN; also at Internet Archive 26 Sept. 2011 23:27:56).

David Herman. 2010b. modules proposal. es-discuss mailing list. 13 May 2010. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2010-May/011122.html (also at Internet Archive 5 June 2014 01:06:35).

David Herman. 2010c. simple modules. es-discuss mailing list. 29 Jan. 2010. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2010-January/010679.html (also at Internet Archive 5 June 2014 01:49:08).

David Herman. 2010d. Strawman: Array Comprehensions. ecmascript.org wiki. 25 June 2010. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:array_comprehensions (BROKEN; also at Internet Archive 15 Feb. 2011 04:49:42).

David Herman. 2010e. Strawman: Module loaders. ecmascript.org wiki. 14 May 2010. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:module_loaders (BROKEN; also at Internet Archive 3 Jan. 2011 06:21:11).

David Herman. 2010f. Strawman: Simple Modules. ecmascript.org wiki. 14 May 2010. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=strawman:simple_modules (BROKEN; also at Internet Archive 18 Sept. 2010 03:53:07).

David Herman. 2011a. dherman/literal-classes.js. GitHub Gist. 1 Nov. 2011. NON-ARCHIVAL https://gist.github.com/dherman/1330478 (also at Internet Archive 14 March 2018 04:11:33).

David Herman. 2011b. ES6 doesn't need opt-in. es-discuss mailing list. 31 Dec. 2011. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2011-December/019112.html (also at Internet Archive 13 Oct. 2012 15:17:15).

David Herman. 2011c. minimal classes. es-discuss mailing list. 27 June 2011. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2011-June/015559.html (also at Internet Archive 1 April 2014 03:50:40).

David Herman. 2011d. Strawman: minimal classes. ecmascript.org wiki. 11 Nov. 2011. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:minimal_classes (BROKEN; also at Internet Archive 25 Dec. 2011 18:47:33).

David Herman. 2011e. Strawman: Pattern Matchings. ecmascript.org wiki. 28 Feb. 2011. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:pattern_matching (BROKEN; also at Internet Archive 6 Dec. 2011 19:55:12).

David Herman. 2012. One JavaScript. Ecma/TC39/2012/005. 18 Jan. 2012. https://www.ecma-international.org/archive/ecmascript/2012/TC39/tc39-2012-005.pdf Presentation at TC39 meeting.

David Herman. 2013a. es6-modules-2013-12-02.pdf. In jorendorff/js-loaders github repository. 2 Dec. 2013. NON-ARCHIVAL https://raw.githubusercontent.com/jorendorff/js-loaders/master/specs/es6-modules-2013-12-02.pdf (also at Internet Archive 3 Sept. 2014 01:02:54).

David Herman. 2013b. Harmony: Module loaders. ecmascript.org wiki. 21 May 2013. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=harmony:module_loaders (BROKEN; also at Internet Archive 27 July 2013 12:49:51).

David Herman. 2014a. A better future for comprehensions. Ecma/TC39/2014/021. 5 June 2014. https://www.ecma-international.org/archive/ecmascript/2014/TC39/tc39-2014-021.pdf Presentation at TC39 meeting.

David Herman. 2014b. Realms API. GitHub Gist. 24 Feb. 2014. NON-ARCHIVAL https://gist.github.com/dherman/7568885 (also at Internet Archive 14 Aug. 2014 22:21:22).

David Herman and Cormac Flanagan. 2007. Status Report: Specifying Javascript with ML. In *Proceedings of the 2007 Workshop on Workshop on ML* (Freiburg, Germany) (*ML '07*). ACM, New York, NY, USA, 47–52. 978-1-59593-676-9 https://doi.org/10.1145/1292535.1292543

David Herman and Yehuda Katz. 2014. Problem: exposing uninitialized built-in objects. Ecma/TC39/2014/045. Sept. 2014. https://ecma-international.org/archive/ecmascript/2014/TC39/tc39-2014-045.pdf

David Herman and Sam Tobin-Hochstadt. 2011. Modules for JavaScript: Simple, Compilable, and Dynamic Libraries on the Web. May 2011. NON-ARCHIVAL http://homes.sice.indiana.edu/samth/js-modules.pdf (also at Internet Archive 25 March 2019 18:38:37). Unpublished paper on author's Web site.

David Herman, Sam Tobin-Hochstadt, and Yahuda Katz. 2013. Modules: Use Cases, Semantics. Presentation to TC39. 12 March 2013. https://www.ecma-international.org/archive/ecmascript/2013/misc/2013misc4.pdf

David Herman, Aaron Tomb, and Cormac Flanagan. 2011. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* 23, 2 (21 Oct), 167. 1573-0557 https://doi.org/10.1007/s10990-011-9066-z

David Herman, Luke Wagner, and Alon Zakai. 2014. asm.js Specification Working Draft. asmjs.org web site. 18 Aug. 2014. NON-ARCHIVAL http://asmjs.org:80/spec/latest/ (also at Internet Archive 28 Aug. 2014 22:43:09).

Ian Hickson. 2004. WHAT open mailing list announcement. whatwg.org web page. June 2004. NON-ARCHIVAL http://whatwg.org/news/start (retrieved 6 May 2019; also at Internet Archive 5 June 2004 21:31:55).

Graydon Hoare. 2010. Rust Language Wiki. Sept. 2010. NON-ARCHIVAL http://github.com/graydon/rust/wiki (also at Internet Archive 7 Oct. 2010 21:47:17).

Darren Hobbs. 2008. Chrome / V8 Javascript performance. Blog post. 2 Sept. 2008. NON-ARCHIVAL http://darrenhobbs.com:80/2008/09/02/chrome-v8-javascript-performance/ (also at Internet Archive 28 Feb. 2012 09:01:03).

Alex Hopmann. 2006. The story of XMLHTTP. Blog post. June 2006. NON-ARCHIVAL http://www.alexhopmann.com:80/xmlhttp.htm (BROKEN; also at Internet Archive 17 June 2006 16:30:47).

Waldemar Horwat. 1998. Revised section 12. Ecma TC39 working document. Sept. 1998. https://www.ecma-international.org/archive/ecmascript/1998/TC39WG/980930-horwat/12.pdf

Waldemar Horwat. 1999a. JavaScript 2.0 February 1999 Draft. Feb. 1999. NON-ARCHIVAL http://mozilla.org:80/js/language/js20-1999-02-18/index.html (also at Internet Archive 16 Aug. 2000 19:45:28).

Waldemar Horwat. 1999b. JavaScript 2.0 Member Lookup. March 1999. NON-ARCHIVAL https://www-archive.mozilla.org/js/language/js20-1999-03-25/member-lookup.html Archived at https://web.archive.org/web/20000823225517/http://mozilla.org:80/js/language/js20-1999-03-25/member-lookup.html Except from March 1999 Draft JavaScript 2.0 proposal.

Waldemar Horwat. 2000. Notes from Aug. 22, 2000 meeting between Waldemar Horwat and Herman Venter. https://www.ecma-international.org/archive/ecmascript/2000/misc/2000misc-2.html

Waldemar Horwat. 2001. JavaScript 2.0: Evolving a Language for Evolving Systems. In *LL1: Lightweight Languages Workshop Proceedings*. MIT Artificial Intelligence Lab (Nov.). NON-ARCHIVAL https://www-archive.mozilla.org/js/language/evolvingJS.pdf (also at Internet Archive 3 Oct. 2019 04:50:18).

Waldemar Horwat. 2003a. ECMAScript 4 Netscape Proposal. (17 Aug 2000, updated 30 June 2003). NON-ARCHIVAL http://www.mozilla.org:80/js/language/es4/index.html (BROKEN; also at Internet Archive 3 Aug. 2003 01:55:58).

Waldemar Horwat. 2003b. ECMAScript Edition 3 Errata. Mozilla.org web page. 9 June 2003. NON-ARCHIVAL http://www.mozilla.org:80/js/language/E262-3-errata.html (also at Internet Archive 14 Aug. 2003 00:12:05).

Waldemar Horwat. 2003c. JavaScript 2.0. Mozilla.org web page. 30 June 2003. NON-ARCHIVAL //www.mozilla.org/js/language/js20.html (BROKEN). Archived at https://web.archive.org/web/20030803020856/http://www.mozilla.org/js/language/js20.html

Waldemar Horwat. 2008a. return when desugaring to closure. es-discuss mailing list. 13 Oct. 2008. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2008-October/007807.html (also at Internet Archive 3 Nov. 2013 21:47:51).

Waldemar Horwat. 2008b. Substatement function definitions. Originally, a message to TC39-TG1 private mailing list. 21 March 2008. https://www.ecma-international.org/archive/ecmascript/2008/misc/FibHist.pdf

Waldemar Horwat. 2009. Full TC39 and SES meeting notes. es-discuss mailing list. 29 Jan. 2009. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2009-January/008711.html (also at Internet Archive 5 June 2014 03:39:18).

Waldemar Horwat. 2010. Sep 30 meeting notes. es-discuss mailing list. 30 Sept. 2010. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2010-September/011880.html (also at Internet Archive 5 June 2014 00:47:33).

Waldemar Horwat et al. 2003. Epimetheus. Mozilla.org web page. Aug. 2003. NON-ARCHIVAL http://www.mozilla.org/js/language/Epimetheus.html (also at Internet Archive 14 Aug. 2003 00:20:31).

Waldemar Horwat et al. 2005. JS2 source code archive. Mozilla source code code repository. NON-ARCHIVAL https://dxr.mozilla.org/js/source/mozilla/js2 (retrieved 11 Feb. 2020; also at Internet Archive 26 Feb. 2020 16:40:05).

Rick Hudson. 2012. River Trail. Ecma/TC39/2012/016. 28 March 2012. https://www.ecma-international.org/archive/ecmascript/2012/TC39/tc39-2012-016.pdf

Rick Hudson. 2014. Strawman: Parallel EcmaScript (River Trail) API. ecmascript.org wiki. 20 Feb. 2014. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:data_parallelism (BROKEN; also at Internet Archive 23 Feb. 2014 03:21:39).

Oliver Hunt. 2009. Problem with Arguments inheriting from Array. es5-discuss mailing list (17 Aug. 20098. 17 Aug. 2009. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es5-discuss/2009-August/003112.html (also at Internet Archive 24 July 2014 03:18:11).

Marco Iansiti and Alan MacCormack. 1997. Developing products on Internet time. *Harvard business review* 75, 5, 108–118. NON-ARCHIVAL https://hbr.org/1997/09/developing-products-on-internet-time (also at Internet Archive 11 Feb. 2020 23:05:50).

IEEE. 2008. *IEEE Standard for Floating-Point Arithmetic. IEEE Std. 754–2008.* IEEE Computer Society, New York, NewYork.

JavaScript Jabber. 2014. The Origin of JavaScript with Brendan Eich. JavaScript Jabber Podcast Episode 124. 3 Sept. 2014. NON-ARCHIVAL http://devchat.tv/js-jabber/124-jsj-the-origin-of-javascript-with-brendan-eich (also at Internet Archive

17 Aug. 2015 16:56:52).

Mike Judge et al. 1996. Beavis and Butt-Head Do America. Movie. Paramount Pictures. The lines from the movie that inspired the naming of Netscape's SpiderMonkey JavaScript engine are transcribed at: https://web.archive.org/web/20190901190842/https://en.wikiquote.org/wiki/Beavis_and_Butt-head_Do_America

Chris Kanaracus. 2007. Mozilla, Microsoft drawing sabers over next JavaScript. *ITWorld.com* (1 Nov.). NON-ARCHIVAL http://www.itworld.com:80/AppDev/4061/071101mozillams/ (BROKEN; also at Internet Archive 3 Nov. 2007 09:04:50).

Kangax. 2010. How ECMAScript 5 still does not allow to subclass an array. Blog post. 15 July 2010. NON-ARCHIVAL http://perfectionkills.com/how-ecmascript-5-still-does-not-allow-to-subclass-an-array/ (also at Internet Archive 20 July 2010 17:05:33).

Yahuda Katz. 2014. JavaScript Modules. Website. NON-ARCHIVAL http://jsmodules.io/ (BROKEN; also at Internet Archive 7 July 2014 22:32:06).

Niall Kennedy. 2008. The story behind Google Chrome. Blog post. 3 Sept. 2008. NON-ARCHIVAL http://www.niallkennedy.com:80/blog/2008/09/google-chrome.html (also at Internet Archive 16 Dec. 2008 02:35:45).

Khronos Group. 2011. *Typed Array Specification, Version 1.0 08 February 2011.* Technical Report. Khronos Group, Beaverton, Oregon USA. NON-ARCHIVAL http://www.khronos.org/registry/typedarray/specs/1.0/ (SUPERSEDED; also at Internet Archive 1 Aug. 2013 06:45:47).

Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. 1991. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, USA. 0262111586

Adam Klein. 2015. An update on Object.observe. es-discuss mailing list. 2 Nov. 2015. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2015-November/044684.html (also at Internet Archive 21 Jan. 2016 21:31:10).

Peter-Paul Koch. 2003. Level 0 DOM. QuirksMode.org Web page. Dec. 2003. NON-ARCHIVAL http://www.quirksmode.org:80/js/dom0.html (also at Internet Archive 5 Dec. 2003 22:39:29).

Kris Kowal. 2009a. CommonJS effort sets JavaScript on path for world domination. *ArsTechnica.com* (1 Dec.). NON-ARCHIVAL https://arstechnica.com/information-technology/2009/12/commonjs-effort-sets-javascript-on-path-for-world-domination/ (also at Internet Archive 23 June 2018 10:14:04).

Kris Kowal. 2009b. Hermetic Evaluation, Modules Strawman. es-discuss mailing list. 30 Sept. 2009. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2009-September/010009.html (also at Internet Archive 5 June 2014 02:36:53).

Kris Kowal and Ihab A.B. Awad. 2009a. Module System for ES-Harmony. Ecma/TC39/2009/011. 11 Feb. 2009. https://www.ecma-international.org/archive/ecmascript/2009/TC39/tc39-2009-011.pdf

Kris Kowal and Ihab A.B. Awad. 2009b. Module System for ES-Harmony (Revised). Online document. Sept. 2009. NON-ARCHIVAL https://docs.google.com/document/pub?id=1hNv1SHh_v_6nD1QBd6pySUJ2U5roAELzW5cqD2tt_WM (also at Internet Archive 22 March 2019 16:08:16).

Paul Krill. 2011. InfoWorld interview: Why Google Dart beats JavaScript. *InfoWorld* (15 Nov.). NON-ARCHIVAL http://www.infoworld.com:80/article/2620869/javascript/infoworld-interview--why-google-dart-beats-javascript.html (also at Internet Archive 5 Oct. 2014 04:38:03).

Pratap Lakshman. 2007a. Discussion: Browser Profile. ecmascript.org wiki. 11 March 2007. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=discussion:browser_profile (BROKEN; also at Internet Archive 3 Nov. 2007 15:16:03).

Pratap Lakshman. 2007b. ES4 Minimalist Proposal - Draft. ecmascript.org wiki. 1 March 2007. NON-ARCHIVAL http://wiki.ecmascript.org/lib/exe/fetch.php?id=discussion%3Abrowser_profile&cache=cache&media=discussion:es4minimalistproposaldraft.rtf (BROKEN; also at Internet Archive 12 Dec. 2007 03:56:42).

Pratap Lakshman. 2007c. JScript Deviations from ES3. ecmascript.org wiki. 24 Sept. 2007. Archived at https://www.ecma-international.org/archive/ecmascript/2007/misc/jscriptdeviationsfromes3.pdf

Pratap Lakshman. 2008. ES3.1: Draft 1. es-discuss mailing list. 28 May 2008. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2008-May/006409.html (also at Internet Archive 5 June 2014 05:11:23).

Pratap Lakshman et al. 2008. ECMAScript 3.1 Draft Specification (28 May 2008). ecmascript.org wiki. 28 May 2008. https://www.ecma-international.org/archive/ecmascript/2008/misc/tc39-es31-draft29may08.pdf

Pratap Lakshman, Douglas Crockford, and Allen Wirfs-Brock. 2007. ES3.1 Proposal Working Draft. ecmascript.org wiki. 15 April 2007. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=es3.1:es3.1_proposal_working_draft (BROKEN; also at Internet Archive 3 Nov. 2007 15:16:13).

Pratap Lakshman and Allen Wirfs-Brock (Eds.). 2009. *ECMA-262, 5th Edition: ECMAScript Language Specification.* Ecma International, Geneva, Switzerland (Dec.). https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262%205th%20edition%20December%202009.pdf

Pratap Lakshman, Allen Wirfs-Brock, et al. 2009. Final draft Standard ECMA-262 5th edition (28 April 2009). Ecma/TC397/2009/025. https://www.ecma-international.org/archive/ecmascript/2009/TC39/tc39-2009-025.pdf

Martin LaMonica. 1995. Group Backs Windows Spec. *InfoWorld* 17, 51 (18 Dec.), 16. NON-ARCHIVAL https://books.google.com/books?id=PDgEAAAAMBAJ&lpg=PA16&dq=Inforworld%20Dec%2018%2C%201995%20ecma&pg=PA16#v=onepage&q&f=false (retrieved 18 April 2019)

Bill Lazar. 1997. Borland's IntraBuilder 1.0. *Softw. Dev.* 5, 1 (Jan.), 15–20. 1070-8588 NON-ARCHIVAL http://www.sdmagazine.com:80/breakrm/products/reviews/s971r1.shtml (BROKEN; also at Internet Archive 15 Aug. 2000 08:46:51).

Steve Leach et al. 2018. The Ginger Project. Project Website. 10 Dec. 2018. NON-ARCHIVAL https://ginger.readthedocs.io/en/latest (also at Internet Archive 10 Dec. 2018 09:50:52).

Russell Leggett. 2012. Finding a "safety syntax" for classes. es-discuss mailing list. 19 March 2012. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2012-March/021430.html (also at Internet Archive 14 Jan. 2013 02:07:22).

Brian Leroux. 2010. wtfjs. Web site. 2 June 2010. NON-ARCHIVAL http://wtfjs.com/ (also at Internet Archive 7 June 2010 02:45:13).

Ted Leung. 2011. JSConf 2011. Blog post. May 2011. NON-ARCHIVAL https://www.sauria.com/blog/2011/05/06/jsconf-2011/ Archived at https://web.archive.org/web/20110512171639/http://www.sauria.com/blog/2011/05/06/jsconf-2011

Clayton Lewis. 1999a. TC39 Chairman's Report to the Co-ordinating Committee 10th November 1999. Ecma/TC39/1999/016. 10 Nov. 1999. https://www.ecma-international.org/archive/ecmascript/1999/TC39/9t39-016.pdf

Clayton Lewis. 1999b. TC39 Chairman's Report to the Co-ordinating Committee 3rd May 1999. Ecma/TC39/1999/008. 3 May 1999. https://www.ecma-international.org/archive/ecmascript/1999/TC39/9t39-008.pdf

Norbert Lindenberg. 2012. *ECMA-402, 1st Edition: ECMAScript Internationalization API Specification.* Ecma International, Geneva, Switzerland (Dec.). http://www.ecma-international.org/ecma-402/1.0/ECMA-402.pdf

C. H. Lindsey. 1993. A History of ALGOL 68. In *The Second ACM SIGPLAN Conference on History of Programming Languages* (Cambridge, Massachusetts, USA) (*HOPL-II*). ACM, New York, NY, USA, 97–132. 0-89791-570-4 https://doi.org/10.1145/154766.155365

Macromedia. 2003. Symbolic Operators : (type). Online reference manual. June 2003. Archived at https://web.archive.org/web/20031212205931if_/http://www.macromedia.com/livedocs/flash/mx2004/main/12_asd21.htm#wp673890 This is the entry in the ActionScript Dictionary for the ActionScript 2 language which describes the use of type declarations.

Macromedia. 2005. *ActionScript 3 Language Specification.* Macromedia, Inc (14 Nov.). https://www.ecma-international.org/archive/ecmascript/2005/misc/as3lang.pdf

Joel Marcey. 2004. TG39 Chairman's Report to: Co-ordinating Committee. Ecma/TC39/2004/018. 8 April 2004. https://www.ecma-international.org/archive/ecmascript/2004/TC39/tc39-2004-018.pdf

Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. 2004. A visual environment for developing context-sensitive term rewriting systems. In *International Conference on Rewriting Techniques and Applications*. Springer, Berlin, Heidelberg, 301–311.

John McCarthy and Michael I Levin. 1965. *LISP 1.5 Programmer's Manual.* M.I.T. Press. 9780262130110

Tom McFarland. 1998. HP ECMAScript comments. email to TC39 working group. 19 May 1998. https://www.ecma-international.org/archive/ecmascript/1998/TC39WG/980519-hpcomma.html

Sebastian McKenzie. 2016. 2015 in review. Blog post. 6 Jan. 2016. NON-ARCHIVAL https://medium.com/@sebmck/2015-in-review-51ac7035e272 (also at Internet Archive 17 Feb. 2016 23:07:33).

Don Melton. 2003. Greetings from the Safari team at Apple Computer. Email message forwarded to kfm-devel mailing list by Dirk Mueller. 7 Jan. 2003. NON-ARCHIVAL https://marc.info/?l=kfm-devel&m=104197092318639&w=2 Archived at https://web.archive.org/save/https://marc.info/?l=kfm-devel&m=104197092318639&w=2

Robinson Meyer. 2014. On the Reign of 'Benevolent Dictators for Life' in Software. *The Atlantic* (17 Jan.). NON-ARCHIVAL https://www.theatlantic.com/technology/archive/2014/01/on-the-reign-of-benevolent-dictators-for-life-in-software/283139/ (also at Internet Archive 19 Jan. 2014 18:40:05).

Microsoft. 1996. Microsoft Internet Explorer 3.0 Beta Now Available. Press Release. 29 May 1996. NON-ARCHIVAL https://news.microsoft.com/1996/05/29/microsoft-internet-explorer-3-0-beta-now-available Archived at https://web.archive.org/web/20141006081714/http://news.microsoft.com/1996/05/29/microsoft-internet-explorer-3-0-beta-now-available//

Microsoft. 1997. Microsoft Design Proposals for the Ecmascript 2.0 Language Specification. Ecma/TC39/1997/032. 10 July 1997. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-032.pdf

Microsoft. 2000a. Microsoft Delivers First .NET Platform Developer Tools for Building Web Services. Press Release. 11 July 2000. NON-ARCHIVAL https://news.microsoft.com/2000/07/11/microsoft-delivers-first-net-platform-developer-tools-for-building-web-services/ (also at Internet Archive 26 Feb. 2020 23:37:13).

Microsoft. 2000b. Microsoft Unveils Vision for Next Generation Internet. Press Release. 22 June 2000. NON-ARCHIVAL https://news.microsoft.com/2000/06/22/microsoft-unveils-vision-for-next-generation-internet/ (also at Internet Archive 19 Dec. 2015 23:07:05).

Microsoft. 2009a. Initial set of ECMAScript 5 conformance tests. Ecma/TC39/2009/030. May 2009. https://www.ecma-international.org/archive/ecmascript/2009/TC39/tc39-2009-030.zip A zipped directory containing the original set of ES5 tests developed by Microsoft.

Microsoft. 2009b. JScript Version Information. Web page. NON-ARCHIVAL http://msdn.microsoft.com:80/en-us/library/s4esdbwz(VS.71).aspx (also at Internet Archive 11 Feb. 2009 11:41:02).

Microsoft. 2016. WPF overview. Web page. Nov. 2016. NON-ARCHIVAL https://docs.microsoft.com/en-us/dotnet/framework/wpf/introduction-to-wpf?view=vs-2019 (also at Internet Archive 11 Feb. 2020 23:28:05).

Microsoft. 2019. www.typescriptlang.org. Website. NON-ARCHIVAL https://www.typescriptlang.org/ (also at Internet Archive 4 April 2019 22:40:13).

Jim Miller. 2007. TC39 Chairman's Report to Coordinating Committee. Ecma/TC39/2007/002. 4 May 2007. https://www.ecma-international.org/archive/ecmascript/2007/TC39/tc39-2007-002.pdf

Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph.D. Dissertation. Johns Hopkins University, Baltimore, MD, USA. Advisor(s) Shapiro, Jonathan S. NON-ARCHIVAL http://erights.org/talks/thesis/markm-thesis.pdf (Some typo fixes and wording improvements). Archived at https://web.archive.org/web/20120205063344/http://www.erights.org/talks/thesis/markm-thesis.pdf

Mark S. Miller. 2008a. Comments regarding: defineProperty/getProperty design sketch. es-discuss mailing list. 23 April 2008. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2008-April/006222.html (also at Internet Archive 5 June 2014 05:22:39).

Mark S. Miller. 2008b. Controlling DontEnum (was: ES4 draft: Object). es-discuss mailing list. 13 March 2008. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2008-March/005759.html (also at Internet Archive 5 June 2014 05:29:45).

Mark S. Miller. 2008c. How much sugar do classes need? es-discuss mailing list. 22 Nov. 2008. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2008-November/008181.html (also at Internet Archive 18 July 2010 15:40:33).

Mark S. Miller. 2008d. Look Ma, no "this". es-discuss mailing list. 19 Aug. 2008. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2008-August/006941.html (also at Internet Archive 2 Nov. 2011 18:08:09).

Mark S. Miller. 2009. Classes as Sugar – old threads revisited (1 of 2). es-discuss mailing list. 30 March 2009. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2009-March/009116.html (also at Internet Archive 5 June 2014 03:26:19).

Mark S. Miller. 2010a. Classes as Sugar. ecmascript.org wiki. 20 Nov. 2010. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:classes_as_sugar (BROKEN; also at Internet Archive 3 Nov. 2011 00:15:25).

Mark S. Miller. 2010b. States and transitions of the attributes of an EcmaScript 5 property. ecmascript.org wiki. 3 Aug. 2010. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=es3.1:attribute_states (BROKEN; also at Internet Archive 14 May 2013 18:20:10).

Mark S. Miller. 2010c. Strawman: Guards. ecmascript.org wiki. 12 Dec. 2010. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:guards (BROKEN; also at Internet Archive 18 Feb. 2011 15:54:02).

Mark S. Miller. 2010d. Syntax for Efficient Traits. ecmascript.org wiki. 22 Sept. 2010. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:syntax_for_efficient_traits (BROKEN; also at Internet Archive 14 Nov. 2011 23:17:12).

Mark S. Miller. 2011a. Classes with Trait Composition. ecmascript.org wiki. 9 May 2011. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:classes_with_trait_composition (BROKEN; also at Internet Archive 19 Aug. 2011 13:58:29).

Mark S. Miller. 2011b. Harmony Classes. ecmascript.org wiki. 9 June 2011. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=harmony:classes (BROKEN; also at Internet Archive 2 July 2011 20:26:24).

Mark S. Miller. 2018. Regarding: Private members break proxies. GitHub issue comment. 17 June 2018. NON-ARCHIVAL https://github.com/tc39/proposal-class-fields/issues/106#issuecomment-397891307 (also at Internet Archive 29 July 2019 16:31:47).

Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. Caja: Safe active content in sanitized JavaScript. Google white paper. July 2008. NON-ARCHIVAL http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf (BROKEN; also at Internet Archive 4 July 2008 19:08:28).

Mark S Miller, Jonathan Shapiro, et al. 2019. eright.org. Website. NON-ARCHIVAL http://erights.org Archived at https://web.archive.org/web/20190808072144/http://erights.org/ Website created January 1999.

Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA. 0262631814

Miniwatts Marketing Group. 2019. Internet Growth Statistics. Internet World Stats website. Nov. 2019. NON-ARCHIVAL https://www.internetworldstats.com/emarketing.htm (also at Internet Archive 3 Feb. 2020 09:42:53).

Eric Miraglia. 2007. A JavaScript Module Pattern. Blog post on Yahoo! User Interface Blog. 12 June 2007. NON-ARCHIVAL https://yuiblog.com/blog/2007/06/12/module-pattern/ (also at Internet Archive 2 July 2007 02:37:33).

Neil Mix. 2008a. Attribute defaults for Object.defineProperty. es-discuss mailing list. 21 Aug. 2008. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2008-August/006979.html (also at Internet Archive 2 Nov. 2011 18:46:47).

Neil Mix. 2008b. Controlling DontEnum (was: ES4 draft: Object). es-discuss mailing list. 13 March 2008. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2008-March/005746.html (also at Internet Archive 5 June 2014 05:29:40).

Gordon E. Moore. 1975. Progress in digital integrated electronics. *IEDM Tech. Digest* 11, 11–13.

Mozilla. 2004. Mozilla Foundation releases the highly anticipated Mozilla Firefox 1.0 web browser. Web page. 9 Nov. 2004. NON-ARCHIVAL http://blog.mozilla.org/press/2004/11/mozilla-foundation-releases-the-highly-anticipated-mozilla-firefox-1-0-web-browser/ (also at Internet Archive 28 Nov. 2013 22:15:35).

Mozilla. 2006a. New in JavaScript 1.7. Web page. Aug. 2006. NON-ARCHIVAL https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/1.7 (also at Internet Archive 23 Aug. 2013 14:56:10).

Mozilla. 2006b. Tamarin Project. Web page. Nov. 2006. NON-ARCHIVAL http://www.mozilla.org:80/projects/tamarin/ (BROKEN; also at Internet Archive 14 Nov. 2006 23:52:48).

Mozilla. 2008a. Core JavaScript 1.5 Reference : __noSuchMethod__. misc reference manual. June 2008. NON-ARCHIVAL http://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Global_Objects/Object/noSuchMethod (also at Internet Archive 24 Aug. 2008 10:46:55).

Mozilla. 2008b. New in JavaScript 1.8. Web page. Aug. 2008. NON-ARCHIVAL http://developer.mozilla.org/en/New_in_JavaScript_1.8 (also at Internet Archive 31 Aug. 2008 00:53:07).

Mozilla Organization. 1998. who we are. early mozilla.org web page. Dec. 1998. NON-ARCHIVAL http://www.mozilla.org/about.html (SUPERSEDED; also at Internet Archive 2 Dec. 1998 15:21:41).

John F Nash. 1950. Equilibrium points in n-person games. *Proceedings of the national academy of sciences* 36, 1, 48–49. https://doi.org/10.1073/pnas.36.1.48

anonymous contributor Netfreak. 2019. SGI Indy—Higher Intellect Vintage Computer Wiki. NON-ARCHIVAL https://wiki.preterhuman.net/SGI_Indy (also at Internet Archive 3 Aug. 2019 22:54:38).

Netscape. 1995a. Netscape to License Sun's Java Programming Language. Press Release. 23 May 1995. NON-ARCHIVAL http://home.netscape.com/newsref/pr/newsrelease25.html (BROKEN; also at Internet Archive 14 June 1997 00:32:24).

Netscape. 1995b. Release Notes: Netscape Naviagor 2.0b1. Web page. Oct. 1995. NON-ARCHIVAL http://www25.netscape.com:80/eng/mozilla/2.0/relnotes/windows-2.0b1.html (BROKEN; also at Internet Archive 19 April 1997 16:05:10).

Netscape. 1996a. Features added after version 1. Web page, part of Netscape 3.0 JavaScript Handbook. Aug. 1996. NON-ARCHIVAL http://home.netscape.com/eng/mozilla/3.0/handbook/javascript/newfunc.htm#1005394 (BROKEN; also at Internet Archive 5 Dec. 1998 01:54:42).

Netscape. 1996b. Navigator Objects. Web page, part of Netscape 2.0 JavaScript Handbook. March 1996. NON-ARCHIVAL http://home.netscape.com/eng/mozilla/2.0/handbook/javascript/navobj.html (BROKEN). Archived at https://web.archive.org/web/19970617232504fw_/http://home.netscape.com/eng/mozilla/2.0/handbook/javascript/navobj.html

Netscape. 1996c. Navigator Scripting. Web page, part of Navigator 2.0 JavaScript Handbook. March 1996. NON-ARCHIVAL http://home.netscape.com:80/eng/mozilla/2.0/handbook/javascript/script.html (BROKEN; also at Internet Archive 17 June 1997 23:24:58).

Netscape. 1996d. Netscape 2.0 JavaScript Handbook. Online manual. March 1996. NON-ARCHIVAL http://home.netscape.com:80/eng/mozilla/2.0/handbook/javascript/index.html (BROKEN; also at Internet Archive 13 June 1997 23:49:17).

Netscape. 1996e. Netscape 3.0 JavaScript Guide. Online manual. Aug. 1996. NON-ARCHIVAL http://home.netscape.com/eng/mozilla/3.0/handbook/javascript/index.html (BROKEN; also at Internet Archive 14 June 1997 04:24:41).

Netscape. 1996f. Netscape Introduces Netscape Enterprise Server 2.0. Press Release. 5 March 1996. NON-ARCHIVAL http://home.netscape.com/newsref/pr/newsrelease99.html (BROKEN; also at Internet Archive 9 July 1997 16:06:19).

Netscape. 1996g. Release Notes: Netscape Navigator for Window, Atlas Preview Release 2. Web page. May 1996. NON-ARCHIVAL http://www20.netscape.com/eng/mozilla/3.0/relnotes/windows-3.0b3.html (BROKEN; also at Internet Archive 12 May 1996 20:39:09). These are the release notes for Navigator 3.0b3.

Netscape. 1997a. JavaScript Security in Communicator 4.x. Online manual. 30 Sept. 1997. NON-ARCHIVAL http://developer.netscape.com:80/docs/manuals/communicator/jssec/index.htm (BROKEN; also at Internet Archive 5 Dec. 1998 07:14:22).

Netscape. 1997b. Netscape Communicator 3.0.2 Source Tree. The Internet Archive Software Collection. Archived at https://archive.org/details/netscape-communicator-3-0-2-source (28 Oct. 2011) This archive consists of a .zip file containing the complete source code tree used to build Netscape Communicator 3.0.2. The the directory named "mocha" contains the source code for the Mocha JavaScript engine.

Netscape. 1997c. What's New in JavaScript for Navigator 4.0: Introduction. Online manual. June 1997. NON-ARCHIVAL http://developer.netscape.com/library/documentation/communicator/jsguide/intro.htm (BROKEN; also at Internet Archive 30 June 1997 09:26:41). Introduction to JavaScript 1.2 new features.

Netscape. 1997d. What's New in JavaScript for Navigator 4.0: Operators. Online manual. June 1997. NON-ARCHIVAL http://developer.netscape.com:80/library/documentation/communicator/jsguide/operator.htm (BROKEN; also at Internet Archive 30 June 1997 09:27:41). Describes new operators in JavaScript 1.2.

Netscape. 2000. Core JavaScript Guide 1.5. Online manual. 28 Sept. 2000. NON-ARCHIVAL http://devedge.netscape.com/library/manuals/2000/javascript/1.5/guide/ (BROKEN; also at Internet Archive 26 Oct. 2002 08:43:19).

Netscape and Sun. 1995. Netscape and Sun Announce Javascript, the Open, Cross-platform Object Scripting Language for Enterprise Networks and the Internet 0. Press Release. 4 Dec. 1995. NON-ARCHIVAL http://home.netscape.com:80/newsref/pr/newsrelease67.html (BROKEN; also at Internet Archive 14 June 1997 00:28:09).

Oscar Nierstrasz. 2000. Identify the Champion, An Organisational Pattern Language for Programme Committees. *Pattern Languages of Program Design* 4, 539–556. NON-ARCHIVAL http://scg.unibe.ch/download/champion/ (also at Internet Archive 28 July 2009 09:42:27).

Shanku Niyogi. 2010. The New JavaScript Engine in Internet Explorer 9. Microsoft IEBlog. 18 March 2010. NON-ARCHIVAL https://blogs.msdn.microsoft.com/ie/2010/03/18/the-new-javascript-engine-in-internet-explorer-9/ (also at Internet Archive 9 March 2016 20:23:17).

Node Foundation. 2018. About the Node.js Foundation. Web page. Aug. 2018. NON-ARCHIVAL https://foundation.nodejs.org/about (SUPERSEDED; also at Internet Archive 13 Aug. 2018 16:47:01). The archived page includes a history of the Node Foundation. In 2020, this URL redirects to a generic page about the OpenJS Foundation.

Node Project. 2009. tinyclouds.org/node. Website. Aug. 2009. NON-ARCHIVAL http://tinyclouds.org:80/node#download (BROKEN; also at Internet Archive 17 Aug. 2009 19:00:07). This page links to source code tarballs for early versions of node.js.

Brent Noorda. 2012. History of Nombas. Web page. April 2012. NON-ARCHIVAL http://www.brent-noorda.com/nombas/history/HistoryOfNombas.html (also at Internet Archive 12 Nov. 2013 23:52:57).

Bob Nystrom. 2011. Harmonious Classes. TC39 working document. May 2011. https://ecma-international.org/archive/ecmascript/2011/misc/2011misc5-May-2011.pdf

Openweb. 2008. Brendan Eich and Arun Ranganathan on ECMAScript Harmony. Openweb Podcast Episode 2. 15 Aug. 2008. NON-ARCHIVAL http://openwebpodcast.com/episode-2-brendan-eich-and-arun-ranganathan-on-ecmascript-harmony (BROKEN; also at Internet Archive 18 Aug. 2008 14:02:42).

Opera. 2013. Opera version history. Web page. 5 Feb. 2013. NON-ARCHIVAL http://www.opera.com:80/docs/history/ (BROKEN; also at Internet Archive 7 March 2013 13:53:54).

Jason Orendorff and David Herman. 2014. js-loaders repository. GitHub repository. 28 Feb. 2014. NON-ARCHIVAL https://github.com/jorendorff/js-loaders (also at Internet Archive 14 Aug. 2014 22:21:22).

John K. Ousterhout. 1997. Scripting: Higher Level Programming for the 21st Century. *IEEE Computer* 31, 23–30.

Seymour Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, Inc., New York, NY, USA. 0-465-04627-4

S Prasanna. 2002. Microsoft's VJ#.Net is made in India. *Express Computer* (29 July). NON-ARCHIVAL http://computer.financialexpress.com/20020729/indnews3.shtml (BROKEN; also at Internet Archive 28 Nov. 2013 17:03:55).

Valerio Proietti. 2006. mootools.net. Website. 24 Oct. 2006. NON-ARCHIVAL http://mootools.net:80/ (SUPERSEDED; also at Internet Archive 24 Oct. 2006 03:25:26).

Dave Raggett. 1998a. ECMA TC39 Working Group meeting 19 Nov 1998. https://www.ecma-international.org/archive/ecmascript/1998/TC39WG/tcn9811.htm

Dave Raggett. 1998b. ECMAScript Proposals. 10 Dec. 1998. https://www.ecma-international.org/archive/ecmascript/1998/TC39WG/981210-dsrdec98.htm

Dave Raggett. 1998c. W3C 'Spice' details. 3 March 1998. https://www.ecma-international.org/archive/ecmascript/1998/TC39WG/980303-spice.htm

Dave Raggett. 1999a. ECMA TC39 meetings 14th October 1999. https://www.ecma-international.org/archive/ecmascript/1999/TC39WG/991014-mod9910.htm

Dave Raggett. 1999b. ECMA TC39 Working Group (technical) meeting notes – 11/12 January 1999. https://www.ecma-international.org/archive/ecmascript/1999/TC39WG/tcn9901.htm

Dave Raggett. 1999c. ECMA TC39 Working Group (technical) meeting notes – 19 February 1999. https://www.ecma-international.org/archive/ecmascript/1999/TC39WG/tcn9902.htm

Dave Raggett. 1999d. ECMAScript Modularity SubGroup Meeting – 25th March 1999. Ecma/TC39/1999/006. https://www.ecma-international.org/archive/ecmascript/1999/TC39/9t39-006.htm

Dave Raggett. 2000. ECMA TC39 meetings 20th January 2000. https://www.ecma-international.org/archive/ecmascript/2000/TC39WG/mins-20jan00.html

Dave Raggett, Chris Dollin, and Steve Leach. 1998. Spice documentation. 27 Sept. 1998. https://www.ecma-international.org/archive/ecmascript/1998/TC39WG/980928-spice-docs/index.html

Lisa Rein. 1997. JavaScript 1.2's evolution as explained by its creator. *Netscape World* 2, 5 (May). NON-ARCHIVAL http://www.netscapeworld.com/netscapeworld/nw-05-1997/nw-05-js12.html (BROKEN; also at Internet Archive 16 June 1997 18:33:20).

John Resig. 2006. jQuery: New Wave Javascript. Website. 3 Feb. 2006. NON-ARCHIVAL http://jquery.com:80/ (SUPERSEDED; also at Internet Archive 3 Feb. 2006 02:57:10). This is the original jQuery website home page.

Reuters. 2000. Microsoft Dominates Browser Battle. *PC World.com* (27 June). NON-ARCHIVAL http://www.pcworld.com:80/news/article.asp?aid=17448 (BROKEN; also at Internet Archive 20 Oct. 2000 22:46:08).

Andreas Rossberg. 2013. Harmony: Refutable Patterns. ecmascript.org wiki. 21 March 2013. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=harmony:refutable_matching (BROKEN; also at Internet Archive 5 June 2013 01:52:14).

Alex Russell et al. 2005. dōjō the browser toolkit. Website. 1 Sept. 2005. NON-ARCHIVAL http://dojotoolkit.org:80/ (SUPERSEDED; also at Internet Archive 1 Sept. 2005 04:43:03). The home page of the early Dojo toolkit website.

Srivats Sampath. 1996. Netscape Application for Associate membership. Ecma/GA/1996/098. 10 Oct. 1996. https://www.ecma-international.org/archive/ecmascript/1996/GA/96-098.pdf

John Schneider, Rok Yu, and Jeff Dyer (Eds.). 2005. *ECMA-357, 2nd Edition: ECMAScript for XML (E4X) Specification.* Ecma International, Geneva, Switzerland (Dec.). https://www.ecma-international.org/publications/files/ECMA-ST-WITHDRAWN/Ecma-357.pdf

William A. Schulze. 2004a. TG1 Convener's Report to TC39s. Ecma/TC39-TG1/2004/006. 24 Sept. 2004. https://www.ecma-international.org/archive/ecmascript/2004/TG1/tc39-tg1-2004-006.pdf

William A. Schulze. 2004b. TG1 of TC39 Intentions. Ecma/TC39-TG1/2004/005. 29 June 2004. https://www.ecma-international.org/archive/ecmascript/2004/TG1/tc39-tg1-2004-005.pdf Presentation to Ecma General Asembly.

Peter Seibel. 2009. *Coders at work: Reflections on the craft of programming.* Apress.

Rawn Shah. 1996. Bending over backward to make JavaScript work on 14 platforms. *JavaWorld* 1, 2 (18 April). NON-ARCHIVAL http://www.javaworld.com/javaworld/jw-04-1996/jw-04-jsinterview.html (also at Internet Archive 4 Jan. 1997 12:22:16).

Remy Sharp. 2010. What is a Polyfill? Blog post. Oct. 2010. NON-ARCHIVAL https://remysharp.com/2010/10/08/what-is-a-polyfill (also at Internet Archive 5 Oct. 2012 20:21:13).

David Singer. 1998. The Future of HTML: A Modest Proposal. Presention at W3C Shapng the Future of HTML Workshop. May 1998. NON-ARCHIVAL https://www.w3.org/MarkUp/future/papers/singer/im-164149.htm (also at Internet Archive 23 Oct. 1999 02:49:10).

Walter R. Smith. 1995. Using a Prototype-based Language for User Interface: The Newton Project's Experience. In *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications* (Austin, Texas, USA) (*OOPSLA '95*). ACM, New York, NY, USA, 61–72. 0-89791-703-0 https://doi.org/10.1145/217838.217844

Maciej Stachowiak. 2007a. Announcing SunSpider 0.9. Webkit.org blog post. 18 Dec. 2007. NON-ARCHIVAL http://webkit.org/blog/152/announcing-sunspider-09/ (also at Internet Archive 21 Dec. 2007 05:57:44).

Maciej Stachowiak. 2007b. RE: Refocus (16 March 2007, 6:21 PM). Message to TC39-TG1 private mailing list. Archived by Ecma International.

Maciej Stachowiak. 2008a. Introducing SquirrelFish Extreme. Webkit.org blog post. 18 Sept. 2008. NON-ARCHIVAL http://webkit.org/blog/214/introducing-squirrelfish-extreme/ (also at Internet Archive 30 Sept. 2008 20:55:51).

Maciej Stachowiak. 2008b. Namespaces as Sugar. ES4-discuss mailing list. 27 May 2008. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2008-May/006395.html (also at Internet Archive 5 June 2014 05:09:24).

Stack Overflow. 2018. Stack Survey 2018 Developer Survey: Most Popular Technologies. Web page. March 2018. NON-ARCHIVAL https://insights.stackoverflow.com/survey/2018/#most-popular-technologies (also at Internet Archive 13 March 2018 07:14:17).

Guy L. Steele, Jr. 1990. *Common LISP: the Language, 2nd Edition.* Elsevier. NON-ARCHIVAL https://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html

Guy L. Steele, Jr. (Ed.). 1997. *ECMA-262: ECMAScript A general purpose cross-platform programming language.* Ecma International, Geneva, Switzerland (June). https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%201st%20edition,%20June%201997.pdf The standard was approved in June 1997 by the Ecma General Assembly, but was not published until September 1997.

Sam Stephenson et al. 2007. Prototype JavaScript Framework. Project Website 25 Jan 2007. 25 Jan. 2007. NON-ARCHIVAL http://www.prototypejs.org:80/ (SUPERSEDED; also at Internet Archive 25 Jan. 2007 18:19:49). This is an early version of the jQuery Prototype framework home page.

Gerald Jay Sussman and Guy L. Steele Jr. 1975. Scheme: An interpreter for extended lambda calculus. https://doi.org/10.1023/A:1010035624696 NON-ARCHIVAL https://dspace.mit.edu/handle/1721.1/5794 AI Memo 349, MIT.

Ankur Taly, Úlfar Erlingsson, John C Mitchell, Mark S Miller, and Jasvir Nagra. 2011. Automated Analysis of Security-Critical JavaScript APIs. In *IEEE Symposium on Security & Privacy (SP)*. IEEE, 363–378. https://ieeexplore.ieee.org/document/5958040/ NON-ARCHIVAL https://storage.googleapis.com/pub-tools-public-publication-data/pdf/37199.pdf NON-ARCHIVAL http://www-cs-students.stanford.edu/~ataly/Papers/sp11.pdf

TC39. 1996. Minutes for the 1st meeting of TC39. Ecma/TC39/1996/004. Dec. 1996. https://www.ecma-international.org/archive/ecmascript/1996/TC39/96-004.pdf

TC39. 1997a. ECMAScript Language Specification, Version 0.12 (with revision marks). Ecma/TC39/1997/017B. 12 March 1997. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-017B.pdf

TC39. 1997b. ECMAScript Language Specification, Version 0.18. Ecma/TC39/1997/028. 2 May 1997. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-028.pdf

TC39. 1997c. ECMAScript Language Specification, Version 0.3. Ecma/TC39/1997/001. 10 Jan. 1997. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-001.pdf

TC39. 1997d. ES1 issue resolution history. 14 April 1997. https://www.ecma-international.org/archive/ecmascript/1997/misc/97misc-1.pdf Extract from ECMA/TC39/97/23.

TC39. 1997e. Minutes for the 2nd meeting of TC39, 14-15 January 1997. Ecma/TC39/1997/007. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-007.pdf
TC39. 1997f. Minutes for the 3rd meeting of TC39, 18-19 March 1997. Ecma/TC39/1997/018. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-018.pdf
TC39. 1997g. Minutes for the 4th meeting of TC39. July 15-16, 2007. Ecma/TC39/1997/030. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-030.pdf
TC39. 1997h. Minutes for the 5th meeting of TC39. September 16-17, 1997. Ecma/TC39/1997/039. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-039.pdf
TC39. 1998a. Disposition of Comments Report for DIS-16262. Ecma/TC39/1998/010. 15 June 1998. https://www.ecma-international.org/archive/ecmascript/1998/TC39/8T39-010.pdf
TC39. 1998b. ECMA comments ISO/IEC DIS 16262, ECMAScript. Ecma/TC39/1998/005. 1 April 1998. https://www.ecma-international.org/archive/ecmascript/1998/TC39/8T39-005.pdf
TC39. 1998c. ECMA TC39 technical meeting - 19 February 1998. https://www.ecma-international.org/archive/ecmascript/1998/TC39WG/tcn9802t.htm
TC39. 1998d. ECMA TC39 Technical Meeting - March 20, 1998. https://www.ecma-international.org/archive/ecmascript/1998/TC39WG/tcn9803.htm
TC39. 1998e. Letter ballot results for DIS 16262. Ecma/TC39/1998/007. 4 May 1998. https://www.ecma-international.org/archive/ecmascript/1998/TC39/8T39-007.pdf
TC39. 1999a. ECMA TC39 meetings 14th-15th November 1999. https://www.ecma-international.org/archive/ecmascript/1999/TC39WG/tcn9911.htm
TC39. 1999b. ECMA TC39 meetings 23-24th September 1999. https://www.ecma-international.org/archive/ecmascript/1999/TC39WG/tcn9909.htm
TC39. 1999c. ECMA TC39 Working Group - Futures list, as of 1999.03.30. Ecma/TC39/1999/004. 30 March 1999. https://www.ecma-international.org/archive/ecmascript/1999/TC39/9t39-004.pdf
TC39. 1999d. ECMA TC39 Working Group - Futures list, as of 1999.11.15. 15 Nov. 1999. https://www.ecma-international.org/archive/ecmascript/1999/TC39WG/991115-futures.htm
TC39. 1999e. ECMAScript Language Specification, Edition 3 Final Draft. (14 Oct. 1999). Ecma/TC39/1999/015. 14 Oct. 1999. https://www.ecma-international.org/archive/ecmascript/1999/TC39/9t39-015.pdf
TC39. 2003. TC39 Email Reflector. From January 2003 through November 2016, TC39 and TC39-TG1 used an email list server operated by the Ecma Secretariat. This was called the TC39-TG1 (and later just TC39) Reflector. It was a private distribution list used for communications among TC39 delegates and with the Ecma Secretariat. Ecma has electronic archives of the messages sent to this list, but as of 2020 they have not been made public. The archived messages were available to the authors of this paper. Requests to access these archives should be submitted to the Ecma Secretary General.
TC39. 2007. ES Wiki. NON-ARCHIVAL http://wiki.ecmascript.org (BROKEN). Archived at https://web.archive.org/web/20150924163114/http://wiki.ecmascript.org/doku.php?id= Originally named "ES4 Wiki" it was renamed to "ES Wiki" in August 2008. From September 2007 through 2015, TC39 used wiki.ecmascript.org to capture and host ECMAScript feature proposals and other materials. That wiki is no longer operational but much of its content is accessible using archive.org. It is necessary to move around the capture timeline to access some of the older material.
TC39. 2007. Minutes for the Ecma TC39-TG1 held in San Francisco, CA on 8-9 November 2007. Ecma/TC39/2007/012. https://www.ecma-international.org/archive/ecmascript/2007/TC39/tc39-2007-012.pdf
TC39. 2008a. Minutes for the 7th meeting of Ecma TC39. Ecma/TC39/2008/105. 20 Nov. 2008. https://www.ecma-international.org/archive/ecmascript/2008/TC39/tc39-2008-105.pdf
TC39. 2008b. Minutes of the 1st meeting of Ecma TC39 Special group on Secure ECMAScript. Ecma/TC39/2008/079. 28 Aug. 2008. https://www.ecma-international.org/archive/ecmascript/2008/TC39/tc39-2008-079.pdf
TC39. 2008c. Minutes of the TC39 ES3.1WG meeting of 26 March 2008. Ecma/TC39/2008/028. https://www.ecma-international.org/archive/ecmascript/2008/TC39/tc39-2008-028.pdf
TC39. 2008d. Minutes of the TC39 ES3.1WG teleconference of 21 February 2008. Ecma/TC39/2008/013. https://ecma-international.org/archive/ecmascript/2008/TC39/tc39-2008-013.pdf
TC39. 2008e. Presentations at the 1st meeting of Ecma TC39 Special group on Secure ECMAScript. Ecma/TC39/2008/086. 28 Aug. 2008. https://www.ecma-international.org/archive/ecmascript/2008/index.html#files-086
TC39. 2008f. Revised Agenda for the 5th meeting of Ecma TC39. Ecma/TC39/2008/054-Rev1. 19 July 2008. https://www.ecma-international.org/archive/ecmascript/2008/TC39/tc39-2008-054-Rev1.pdf
TC39. 2008g. Revised Minutes of the 5th meeting of Ecma TC39 23-25 July 2008. Ecma/TC39/2008/067-Rev1. 30 Sept. 2008. https://www.ecma-international.org/archive/ecmascript/2008/TC39/tc39-2008-067-Rev1.pdf
TC39. 2009a. Minutes for the 11th meeting of Ecma TC39 29-30 July 2009. Ecma/TC39/2009/037-Rev1. https://www.ecma-international.org/archive/ecmascript/2009/TC39/tc39-2009-037-Rev1.pdf

TC39. 2009b. Minutes for the 12th meeting of Ecma TC39 23-24 September 2009. Ecma/TC39/2009/045. https://www.ecma-international.org/archive/ecmascript/2009/TC39/tc39-2009-045.pdf
TC39. 2009c. Minutes for the 8th meeting of Ecma TC39 28-29 January 2009. Ecma/TC39/2009/008-Rev1. https://www.ecma-international.org/archive/ecmascript/2009/TC39/tc39-2009-008-Rev1.pdf
TC39. 2009d. Minutes for the 9th meeting of Ecma TC39 25-26 March 2009. Ecma/TC39/2009/022. https://www.ecma-international.org/archive/ecmascript/2009/TC39/tc39-2009-022.pdf
TC39. 2010. Minutes for the 16th meeting of Ecma TC39 24-25 May 2010. Ecma/TC39/2010/028. https://www.ecma-international.org/archive/ecmascript/2010/TC39/tc39-2010-028.pdf
TC39. 2011a. Draft Minutes for the 23rd meeting of Ecma TC39 27-28 July 2011. Ecma/TC39/2011/037. https://www.ecma-international.org/archive/ecmascript/2011/TC39/tc39-2011-037.pdf
TC39. 2011b. Minutes for the 22nd meeting of Ecma TC39 24-26 May 2011. Ecma/TC39/2011/028. https://www.ecma-international.org/archive/ecmascript/2011/TC39/tc39-2011-028.pdf
TC39. 2012a. Minutes for the 27th meeting of Ecma TC39, Revision 2. Ecma/TC39/2012/020-rev2. 30 March 2012. https://www.ecma-international.org/archive/ecmascript/2012/TC39/tc39-2012-020-Rev2.pdf
TC39. 2012b. Minutes for the 28th meeting of Ecma TC39. Ecma/TC39/2012/034. 23 May 2012. https://www.ecma-international.org/archive/ecmascript/2012/TC39/tc39-2012-034.pdf
TC39. 2012c. Minutes for the 29th meeting of Ecma TC39. Ecma/TC39/2012/056. 24 July 2012. https://www.ecma-international.org/archive/ecmascript/2012/TC39/tc39-2012-056.pdf
TC39. 2013a. March 14, 2013 Meeting Notes. https://www.ecma-international.org/archive/ecmascript/2013/notes/2013-03/mar-14.html
TC39. 2013b. Minutes for the 32nd meeting of Ecma TC39 January 29-31, 2013. Ecma/TC39/2013/009. https://www.ecma-international.org/archive/ecmascript/2013/TC39/tc39-2013-009.pdf
TC39. 2013c. Minutes for the 36th meeting of Ecma TC39. Ecma/TC39/2013/055. 17 Sept. 2013. https://www.ecma-international.org/archive/ecmascript/2013/TC39/tc39-2013-055.pdf
TC39. 2014a. July 30, 2014 Meeting Notes. https://web.archive.org/web/20190820220952/https://tc39.es/tc39-notes/2014-07_jul-30.html
TC39. 2014b. Minutes for the 42th meeting of Ecma TC39. Ecma/TC39/2014/051. 25 Sept. 2014. https://www.ecma-international.org/archive/ecmascript/2014/TC39/tc39-2014-051.pdf
TC39. 2015a. January 27, 2015 Meeting Notes. https://www.ecma-international.org/archive/ecmascript/2015/notes/2015-01/jan-27.html
TC39. 2015b. Minutes for the 45th meeting of Ecma TC39. Ecma/TC39/2015/031. 17 March 2015. https://www.ecma-international.org/archive/ecmascript/2015/TC39/tc39-2015-031.pdf
TC39 et al. 2006. The es-discuss Archives. Email forum hosted by mozilla.org. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/ (also at Internet Archive 6 Nov. 2017 17:08:35). Originally named es4-discuss. Renamed in August 2008.
TC39 et al. 2008. The es5-discuss Archives. Email forum hosted by mozilla.org. April 2008. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es5-discuss/ (also at Internet Archive 14 Jan. 2019 10:49:03). Originally named es3.1-discuss. Renamed in March 2009.
TC39 et al. 2016. TC39 Bugzilla Archive, 2011–2015. NON-ARCHIVAL https://tc39.es/archives/bugzilla/ This is a formatted listing of a data export from the original TC39 Bugzilla server.
TC39 ES4. 2006a. Catchall proposal. ecmascript.org wiki. Nov. 2006. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=proposals:catchalls (BROKEN; also at Internet Archive 20 Oct. 2007 08:25:39).
TC39 ES4. 2006b. Clarification: Type System. ES4 Wiki. 23 Aug. 2006. NON-ARCHIVAL http://developer.mozilla.org:80/es4/clarification/type_system.html (BROKEN; also at Internet Archive 14 Jan. 2007 06:19:15).
TC39 ES4. 2006c. Expression Closures proposal. ecmascript.org wiki. Sept. 2006. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=proposals:expression_closures (BROKEN; also at Internet Archive 20 Oct. 2007 08:26:04).
TC39 ES4. 2006d. Proposals: Structural types and typing of initializers. ES4 Wiki. 17 June 2006. NON-ARCHIVAL http://developer.mozilla.org:80/es4/proposals/structural_types_and_typing_of_initializers.html (BROKEN; also at Internet Archive 17 June 2006 07:44:06).
TC39 ES4. 2007a. Clarification: Formal Type System. ES4 Wiki. 16 Jan. 2007. NON-ARCHIVAL http://developer.mozilla.org:80/es4/clarification/formal_type_system.html (BROKEN; also at Internet Archive 16 Jan. 2007 07:21:22).
TC39 ES4. 2007b. Clarification: Type System. ecmascript.org wiki. 7 July 2007. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=clarification:type_system (BROKEN; also at Internet Archive 8 Nov. 2007 07:52:29).
TC39 ES4. 2007c. ECMAScript Documentation. www.ecmascript.org website. 27 Oct. 2007. NON-ARCHIVAL http://www.ecmascript.org:80/docs.php (BROKEN; also at Internet Archive 27 Oct. 2007 09:38:37).
TC39 ES4. 2007d. ES4 Pre Release M0 Source. www.ecmascript.org website. 8 June 2007. NON-ARCHIVAL http://www.ecmascript.org/files/es4-pre-release.M0.source.tar.gz (BROKEN; also at Internet Archive 31 Oct. 2007 13:08:00). This is the

first public source code release of the $ES4_2$ reference implementation. The final source code as of the termination of the project in July 2008 is at NON-ARCHIVAL https://github.com/dherman/es4.

TC39 ES4. 2007e. Proposals for modifying the spec. ecmascript.org wiki. 29 Sept. 2007. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=proposals:proposals (BROKEN; also at Internet Archive 20 Oct. 2007 01:50:54).

TC39 ES4. 2007f. Proposals: Inactive. ecmascript.org wiki. 29 Sept. 2007. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=proposals:inactive (BROKEN; also at Internet Archive 20 Oct. 2007 08:26:35).

TC39 ES4. 2007g. Public snapshot of TC39-TG1's private ES4 wiki. Jan. 2007. NON-ARCHIVAL http://developer.mozilla.org/es4/ (BROKEN; also at Internet Archive 3 Jan. 2007 22:37:12).

TC39 Harmony. 2008. Strawman Proposals. ecmascript.org wiki. 21 Nov. 2008. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=strawman:strawman (BROKEN; also at Internet Archive 22 Dec. 2008 19:08:52).

TC39 Harmony. 2009. Strawman Proposals. ecmascript.org wiki. 3 Aug. 2009. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=strawman:strawman (also at Internet Archive 18 Aug. 2009 15:32:07).

TC39 Harmony. 2010a. Deferred Proposals. ecmascript.org wiki. 23 Nov. 2010. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:deferred (BROKEN; also at Internet Archive 6 Dec. 2011 19:08:02).

TC39 Harmony. 2010b. Strawman Proposals. ecmascript.org wiki. 22 Dec. 2010. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=strawman:strawman (BROKEN; also at Internet Archive 31 Dec. 2010 08:39:07).

TC39 Harmony. 2010c. Strawman: Shorter function syntax. ecmascript.org wiki. May 2010. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:shorter_function_syntax (BROKEN; also at Internet Archive 22 Jan. 2011 01:14:36).

TC39 Harmony. 2011a. Harmony Proposals. ecmascript.org wiki. 23 March 2011. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=harmony:proposals (BROKEN; also at Internet Archive 24 April 2011 16:23:26).

TC39 Harmony. 2011b. Harmony Proposals. ecmascript.org wiki. 1 June 2011. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=harmony:proposals (BROKEN; also at Internet Archive 25 June 2011 00:55:57).

TC39 Harmony. 2011c. Strawman Proposals. ecmascript.org wiki. 28 April 2011. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=strawman:strawman (BROKEN; also at Internet Archive 1 May 2011 19:23:03).

TC39 Harmony. 2014. Harmony Proposals. ecmascript.org wiki. 30 Jan. 2014. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=harmony:proposals (BROKEN; also at Internet Archive 14 Feb. 2014 12:05:27).

TC39 Harmony. 2015. Harmony: Specification Drafts. ecmascript.org wiki. 17 April 2015. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=harmony:specification_drafts (BROKEN; also at Internet Archive 19 April 2015 08:03:39).

TC39-TG1. 2005. Minutes of Ecma TC39-TG1 September 22, 2005. Ecma/TC39-TG1/2005/006. https://www.ecma-international.org/archive/ecmascript/2005/TG1/tc39-tg1-2005-006.pdf

TC39-TG1. 2006a. Minutes of Ecma TC39-TG1 April 21, 2006. Ecma/TC39-TG1/2006/020. https://www.ecma-international.org/archive/ecmascript/2006/TG1/tc39-tg1-2006-020.pdf

TC39-TG1. 2006b. Minutes of Ecma TC39-TG1 February 16, 2006. Ecma/TC39-TG1/2006/011. https://www.ecma-international.org/archive/ecmascript/2006/TG1/tc39-tg1-2006-011.pdf

TC39-TG1. 2006c. Minutes of Ecma TC39-TG1 July 27-28, 2006. Ecma/TC39-TG1/2006/032. https://www.ecma-international.org/archive/ecmascript/2006/TG1/tc39-tg1-2006-032.pdf

TC39-TG1. 2006d. Minutes of Ecma TC39-TG1 March 16, 2006. Ecma/TC39-TG1/2006/015. https://www.ecma-international.org/archive/ecmascript/2006/TG1/tc39-tg1-2006-015.pdf

TC39-TG1. 2006e. Minutes of Ecma TC39-TG1 October 19-20, 2006. Ecma/TC39-TG1/2006/041. https://www.ecma-international.org/archive/ecmascript/2006/TG1/tc39-tg1-2006-041.pdf

TC39-TG1. 2007a. Minutes of Ecma TC39-TG1 April 18-20, 2007. Ecma/TC39-TG1/2007/017. https://www.ecma-international.org/archive/ecmascript/2007/TG1/tc39-tg1-2007-017.pdf

TC39-TG1. 2007b. Minutes of Ecma TC39-TG1 June 21-22, 2007. Ecma/TC39-TG1/2007/025. https://www.ecma-international.org/archive/ecmascript/2007/TG1/tc39-tg1-2007-025.pdf

TC39-TG1. 2007c. Minutes of Ecma TC39-TG1 March 21-23, 2007. Ecma/TC39-TG1/2007/013. https://www.ecma-international.org/archive/ecmascript/2007/TG1/tc39-tg1-2007-013.pdf

TC39-TG1. 2007d. Minutes of Ecma TC39-TG1 September 27-28, 2007. Ecma/TC39-TG1/2007/036. https://www.ecma-international.org/archive/ecmascript/2007/TG1/tc39-tg1-2007-036.pdf

R.D. Tennent. 1981. *Principles of programming languages.* Prentice/Hall International. 9780137098736 80024271

Brian Terlson. 2012. Real World Func Decl in Block Scope Breakages. es-discuss mailing list. 26 Dec. 2012. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2012-December/027419.html (also at Internet Archive 4 June 2014 15:53:17).

Sam Tobin-Hochstadt. 2010. simple modules. es-discuss mailing list. 4 Feb. 2010. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2010-February/010782.html (also at Internet Archive 5 June 2014 01:27:32).

Sam Tobin-Hochstadt and David Herman. 2010. Simple Modules. Ecma/TC39/2010/017. 24 March 2010. https://www.ecma-international.org/archive/ecmascript/2010/TC39/tc39-2010-017.pdf Presentation at TC39 meeting.

Traceur Project. 2011a. traceur-compiler project. Google Code Project. 5 May 2011. NON-ARCHIVAL http://code.google.com/p/traceur-compiler/ (SUPERSEDED; also at Internet Archive 5 May 2011 01:15:49). The Traceur GitHub repository

NON-ARCHIVAL https://github.com/google/traceur-compiler

Traceur Project. 2011b. Traceur Language Features - Classes. Google Code Project. 4 May 2011. NON-ARCHIVAL http://code.google.com/p/traceur-compiler/wiki/LanguageFeatures#Classes (SUPERSEDED; also at Internet Archive 7 May 2011 11:59:13).

David Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications* (Orlando, Florida, USA) (*OOPSLA '87*). ACM, New York, NY, USA, 227–242. 0-89791-247-0 https://doi.org/10.1145/38765.38828

Tom Van Cutsem. 2009. Catch-all proposal based on proxies. es-discuss mailing list. 7 Dec. 2009. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2009-December/010250.html (also at Internet Archive 6 Aug. 2013 07:15:18).

Tom Van Cutsem. 2011. Direct proxies strawman. es-discuss mailing list. 17 Oct. 2011. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2011-October/017466.html (also at Internet Archive 4 June 2014 20:08:02).

Tom Van Cutsem. 2013. Notification Proxies. Blog Post. 22 May 2013. NON-ARCHIVAL http://tvcutsem.github.io/notification-proxies (also at Internet Archive 22 Jan. 2017 02:37:05).

Tom Van Cutsem and Mark S. Miller. 2010a. Catch-all Proxies. ecmascript.org wiki. March 2010. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=harmony:proxies (BROKEN; also at Internet Archive 24 March 2010 21:47:41).

Tom Van Cutsem and Mark S. Miller. 2010b. Proxies: Design Principles for Robust Object-oriented Intercession APIs. In *Proceedings of the 6th Symposium on Dynamic Languages* (Reno/Tahoe, Nevada, USA) (*DLS '10*). ACM, New York, NY, USA, 59–72. 978-1-4503-0405-4 https://doi.org/10.1145/1869631.1869638

Tom Van Cutsem and Mark S. Miller. 2010c. Proxies Strawman Proposal. Ecma/TC39/2010/007. 27 Jan. 2010. https://www.ecma-international.org/archive/ecmascript/2010/TC39/tc39-2010-007.pdf Presentation at TC39 meeting.

Tom Van Cutsem and Mark S. Miller. 2011a. Direct Proxy Spec. ecmascript.org wiki. 23 Dec. 2011. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=harmony:proxies_spec (BROKEN; also at Internet Archive 14 May 2012 05:55:57).

Tom Van Cutsem and Mark S. Miller. 2011b. Strawman: Direct Proxies. ecmascript.org wiki. 23 Nov. 2011. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:direct_proxies (BROKEN; also at Internet Archive 13 July 2012 16:15:48).

Tom Van Cutsem and Mark S. Miller. 2011c. Traits.Js: Robust Object Composition and High-integrity Objects for Ecmascript 5. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients* (Portland, Oregon, USA) (*PLASTIC '11*). ACM, New York, NY, USA, 1–8. 978-1-4503-1171-7 https://doi.org/10.1145/2093328.2093330

Tom Van Cutsem and Mark S. Miller. 2012. Proposal: Direct Proxies. ecmascript.org wiki. 22 March 2012. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=harmony:direct_proxies (BROKEN; also at Internet Archive 23 April 2012 15:01:31).

Tom Van Cutsem and Mark S. Miller. 2013. Trustworthy Proxies: Virtualizing Objects with Invariants. In *Proceedings of the 27th European Conference on Object-Oriented Programming* (Montpellier, France) (*ECOOP'13*). Springer-Verlag, Berlin, Heidelberg, 154–178. 978-3-642-39037-1 https://doi.org/10.1007/978-3-642-39038-8_7

Jim Van Eaton. 2005. Outlook Web Access - A catalyst for web evolution. Microsoft Exchange Team Blog. 21 June 2005. NON-ARCHIVAL http://msexchangeteam.com/archive/2005/06/21/406646.aspx (BROKEN; also at Internet Archive 23 June 2006 17:50:04).

Markku Vartiainen (Ed.). 2001. *ECMA-327, ECMAScript 3rd Edition Compact Profile.* Ecma International, Geneva, Switzerland (June). https://www.ecma-international.org/publications/files/ECMA-ST-WITHDRAWN/Ecma-327.pdf

Herman Venter. 1998a. instanceof proposal. Ecma TC39 working document. 5 March 1998. https://www.ecma-international.org/archive/ecmascript/1998/TC39WG/980305-instance.pdf

Herman Venter. 1998b. Revised section 12. Ecma TC39 working document. 5 March 1998. https://www.ecma-international.org/archive/ecmascript/1998/TC39WG/980305-labelled.pdf

Herman Venter. 1998c. Updates for section 12. Ecma TC39 working document. 22 April 1998. https://www.ecma-international.org/archive/ecmascript/1998/TC39WG/980430-label3.pdf

Eelco Visser. 2001. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications* (*RTA '01*). Springer-Verlag, Berlin, Heidelberg, 357–362. 3-540-42117-3 http://dl.acm.org/citation.cfm?id=647200.718711

W3C. 1998. Shaping the Future of HTML. W3C web pages. May 1998. NON-ARCHIVAL https://www.w3.org/MarkUp/future/ (also at Internet Archive 3 July 1998 15:57:23). Record of W3C workshop held 4–5 May 1998.

Richard Wagner (Ed.). 1999. *ECMA-290: ECMAScript Components Specification.* Ecma International, Geneva, Switzerland (June). https://www.ecma-international.org/publications/files/ECMA-ST-WITHDRAWN/ECMA-290.pdf

Richard Wagner and Chip Shapley. 1998. ECMAScript Components Technical Specification. (25 June 1998). Ecma/TC39/1998/011. 25 June 1998. https://www.ecma-international.org/archive/ecmascript/1998/TC39/8T39-011.pdf

Joe Walker. 2018. The Brave Cookie Monster—Brendan Eich. The Jolly Swagman Podcast Episode 50. 29 May 2018. Audio recording NON-ARCHIVAL https://content.blubrry.com/thejollyswagmen/JollySwagmen050-BrendanEich-FINAL.mp3 (also at Internet Archive 12 Feb. 2020 16:44:00). Show Notes NON-ARCHIVAL https://josephnoelwalker.com/50-the-brave-cookie-monster-brendan-eich/

Rafael Weinstein. 2012. Harmony: Observe. ecmascript.org wiki. 16 March 2012. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=harmony:observe (BROKEN; also at Internet Archive 16 Nov. 2012 14:24:41).

Rafael Weinstein and Dmitry Lomov. 2013. Post-ES6 Spec Process. Presentation to TC39. 18 Sept. 2013. NON-ARCHIVAL http://slides.com/rafaelweinstein/tc39-process/# (also at Internet Archive 24 July 2014 03:33:36).

Rafael Weinstein and Allen Wirfs-Brock. 2013. TC-39 Process. Ecma/TC39/2013/062. Nov. 2013. https://www.ecma-international.org/archive/ecmascript/2013/TC39/tc39-2013-062.pdf

Robert Welland et al. 1996. The JScript Language Specification, Version 0.1. Ecma/TC39/1996/005. Nov. 1996. https://www.ecma-international.org/archive/ecmascript/1996/TC39/96-005.pdf

Robert Welland, Shon Katzenberger, and Peter Kukol. 2018. Oral history of members of original Microsoft JScript development team. 22 March 2018. NON-ARCHIVAL http://www.wirfs-brock.com/allen/files/jshistory/JScriptInterview.mp3 (also at Internet Archive 7 March 2020 17:37:32). Audio recording, duration 52:44. Interviewer Allen Wirfs-Brock.

Wikinews. 2007. Wikinews interviews World Wide Web co-inventor Robert Cailliau. Wikinew website. 16 Aug. 2007. NON-ARCHIVAL http://en.wikinews.org/wiki/Wikinews_interviews_World_Wide_Web_co-inventor_Robert_Cailliau (also at Internet Archive 5 Dec. 2007 22:04:09).

Wikipedia. 2019. Embrace, extend, and extinguish—Wikipedia, The Free Encyclopedia. NON-ARCHIVAL https://en.wikipedia.org/wiki/Embrace,_extend,_and_extinguish (retrieved 25 July 2019)

Scott Wiltamuth. 1997a. 1/15 working group meeting notes. Ecma/TC39/1997/005. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-005.pdf

Scott Wiltamuth. 1997b. 1/24 working group notes. Ecma/TC39/1997/008. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-008.pdf

Scott Wiltamuth. 1997c. 4/16 Working group meeting notes. Ecma/TC39/1997/025. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-025.pdf

Scott Wiltamuth. 1997d. Notes from the 1/31 working group meeting. Ecma/TC39/1997/009. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-009.pdf

Scott Wiltamuth. 1997e. Notes from the 2/11 conference call. Ecma/TC39/1997/010. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-010.pdf

Scott Wiltamuth. 1997f. Notes from the 2/14 working group meeting. Ecma/TC39/1997/015. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-015.pdf

Scott Wiltamuth. 1997g. Notes from the 2/28 working group meeting. Ecma/TC39/1997/012. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-012.pdf

Scott Wiltamuth. 1997h. Notes from the 3/14 working group meeting. Ecma/TC39/1997/024. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-024.pdf

Scott Wiltamuth. 1997i. Notes from the 3/18 TC39 technical meeting. Ecma/TC39/1997/026. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-026.pdf

Scott Wiltamuth. 1997j. Selecting a new name to replace "ECMAScript". Ecma/TC39/1997/002. 14 Jan. 1997. https://www.ecma-international.org/archive/ecmascript/1997/TC39/97-002.pdf

Nick Wingfield. 1995. Microsoft storms the Web. *InfoWorld* 17, 50 (11 Dec.), 1. NON-ARCHIVAL https://books.google.com/books?id=QjgEAAAAMBAJ&lpg=PP1&dq=Inforworld%2520Dec%252011%252C%25201995&pg=PP3#v=onepage&q&f=false (also at Internet Archive 27 Feb. 2020 21:15:10).

Allen Wirfs-Brock. 2007a. Implementation Loopholes In ECMAScript, 3rd Edition. ecmascript.org wiki. Aug. 2007. https://www.ecma-international.org/archive/ecmascript/2007/misc/es3_implementation_loopholes_annotated.pdf

Allen Wirfs-Brock. 2007b. Mozilla Extensions to ECMAScript, 3rd Edition. ecmascript.org wiki. Aug. 2007. https://www.ecma-international.org/archive/ecmascript/2007/misc/mozilla_javascript_extensions.pdf

Allen Wirfs-Brock. 2007c. Re: ECMAScript 4 Language Overview White Paper (23 Oct. 2007, 5:47 AM). Message to TC39-TG1 private mailing list. Archived by Ecma International.

Allen Wirfs-Brock. 2008. Proposed ECMAScript 3.1 Static Object Functions: Use Cases and Rationale. ecmascript.org wiki. 26 Aug. 2008. https://www.ecma-international.org/archive/ecmascript/2008/misc/rationale_for_es3_1_static_object_methodsaug26.pdf

Allen Wirfs-Brock. 2009. definitional interpreter for ECMAScript 5 implemented using ECMAScript. Ecma/TC39/2009/052. Oct. 2009. https://www.ecma-international.org/archive/ecmascript/2009/TC39/tc39-2009-052.pdf

Allen Wirfs-Brock. 2010. simple modules. es-discuss mailing list. 3 Feb. 2010. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2010-February/010776.html (also at Internet Archive 5 June 2014 01:27:16).

Allen Wirfs-Brock. 2011a. Declarative Object and Class Abstractions Based Upon Extended Object Initialisers. Ecma/TC39/2011/019. 23 March 2011. https://www.ecma-international.org/archive/ecmascript/2011/TC39/tc39-2011-019.pdf

Allen Wirfs-Brock (Ed.). 2011b. *ECMA-262, Edition 5.1: ECMAScript Language Specification.* Ecma International, Geneva, Switzerland (June). https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262%205.

1%20edition%20June%202011.pdf

Allen Wirfs-Brock. 2011c. Other Object Initialiser Property Modifiers. ecmascript.org wiki. 23 March 2011. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:obj_initialiser_const (BROKEN; also at Internet Archive 15 May 2013 09:27:24).

Allen Wirfs-Brock. 2011d. Strawman: Declarative Object and Class Abstractions Based Upon Extended Object Initialisers. ecmascript.org wiki. March 2011. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:object_initialiser_extensions (BROKEN; also at Internet Archive 22 Aug. 2011 01:33:07).

Allen Wirfs-Brock. 2012a. Block Lambdas: break and continue. es-discuss mailing list. 14 Jan. 2012. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2012-January/019520.html (also at Internet Archive 4 June 2014 18:55:57).

Allen Wirfs-Brock. 2012b. ES6 Max-min class semantics with TC39 decision annotations. Ecma/TC39/2012/054 with annotations. 26 July 2012. https://www.ecma-international.org/archive/ecmascript/2012/misc/2012misc6.pdf

Allen Wirfs-Brock. 2012c. ES6 Subclassing Built-ins. Ecma/TC39/2012/misc5. July 2012. https://www.ecma-international.org/archive/ecmascript/2012/misc/2012misc5.pdf Presentation slides at July 2012 TC39 meeting.

Allen Wirfs-Brock. 2012d. Strawman: maximally minimal classes. ecmascript.org wiki. 25 March 2012. NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=strawman:maximally_minimal_classes (BROKEN; also at Internet Archive 26 April 2012 12:54:28).

Allen Wirfs-Brock. 2012e. "Subclassing" Built-in Constructors. ecmascript.org wiki. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=strawman:subclassable-builtins (BROKEN; also at Internet Archive 15 May 2013 08:39:50).

Allen Wirfs-Brock. 2013. Making Built-in and Exotic Objects Subclassable. Ecma/TC39/2013/misc1. 29 Jan. 2013. https://ecma-international.org/archive/ecmascript/2013/misc/2013misc1.pdf Presentation slides at January 2013 TC39 meeting.

Allen Wirfs-Brock (Ed.). 2015a. *ECMA-262, 6th Edition: ECMAScript 2015 Language Specification.* Ecma International, Geneva, Switzerland (June). https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262%206th%20edition%20June%202015.pdf

Allen Wirfs-Brock. 2015b. ES6 super [[construct]] proposal. Ecma/TC39/2015/misc1. Jan. 2015. https://www.ecma-international.org/archive/ecmascript/2015/misc/2015misc1.html

Allen Wirfs-Brock et al. 2007. Position Statement to TC39-TG1 Regarding the Evolution of ECMAScript. Ecma/TC39-TG1/2007/042. 7 Nov. 2007. https://www.ecma-international.org/archive/ecmascript/2007/TG1/tc39-tg1-2007-042.pdf

Allen Wirfs-Brock et al. 2011a. Draft Specification for ES.next: July 12, 2011. ecmascript.org wiki. 12 July 2011. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts (BROKEN; also at Internet Archive 13 Aug. 2011 14:59:41).

Allen Wirfs-Brock et al. 2011b. Draft Standard ECMA-262 6th Edition, Rev 1. Ecma/TC39/2011/032. 11 July 2011. https://www.ecma-international.org/archive/ecmascript/2011/TC39/tc39-2011-032.pdf

Allen Wirfs-Brock et al. 2012a. Draft Specification for ES.next: Feb. 27, 2012. ecmascript.org wiki. 27 Feb. 2012. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts (BROKEN; also at Internet Archive 4 May 2012 04:11:39).

Allen Wirfs-Brock et al. 2012b. Draft Specification for ES.next: Sept. 27, 2012. ecmascript.org wiki. 27 Sept. 2012. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts (BROKEN; also at Internet Archive 13 Jan. 2013 13:46:59).

Allen Wirfs-Brock et al. 2012c. Sixth draft, Standard ECMA-262 6th edition. Ecma/TC39/2012/071. 27 Sept. 2012. https://www.ecma-international.org/archive/ecmascript/2012/TC39/tc39-2012-071.pdf

Allen Wirfs-Brock et al. 2014a. Draft Specification for ES.next: January 20, 2014 Draft Rev 22. ecmascript.org wiki. 20 Jan. 2014. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts#january_20_2014_draft_rev_22 (BROKEN; also at Internet Archive 23 Jan. 2014 03:43:28).

Allen Wirfs-Brock et al. 2014b. Draft Specification for ES.next: October 14, 2014 Draft Rev 28. ecmascript.org wiki. 14 Oct. 2014. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts#october_14_2014_draft_rev_28 (BROKEN; also at Internet Archive 28 Oct. 2014 07:21:19).

Allen Wirfs-Brock et al. 2014c. Presentation: Instantiation Reform. Ecma/TC39/2014/032. 26 July 2014. https://ecma-international.org/archive/ecmascript/2014/TC39/tc39-2014-032.pdf

Allen Wirfs-Brock et al. 2014d. Presentation: Object Instantiation Redo. Ecma/TC39/2014/046. Sept. 2014. https://ecma-international.org/archive/ecmascript/2014/TC39/tc39-2014-046.pdf

Allen Wirfs-Brock et al. 2015a. Draft Specification for ES.next: April 14, 2015 Draft Rev 38. ecmascript.org wiki. 14 April 2015. Originally at NON-ARCHIVAL http://wiki.ecmascript.org:80/doku.php?id=harmony:specification_drafts#final_draft (BROKEN; also at Internet Archive 19 April 2015 08:03:39).

Allen Wirfs-Brock et al. 2015b. Draft Specification for ES.next: March 17, 2015 Draft Rev 36. ecmascript.org wiki. 17 March 2015. NON-ARCHIVAL http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts#march_17_2015_rev_36_release_candidate_3 (BROKEN; also at Internet Archive 24 March 2015 06:04:22).

Allen Wirfs-Brock et al. 2015c. Final draft Standard ECMA-262 6th Edition. Ecma/TC39/2015/030. April 2015. https://www.ecma-international.org/archive/ecmascript/2015/TC39/tc39-2015-030.pdf

Allen Wirfs-Brock and Douglas Crockford. 2007. Notes from 8/16/07 Allen Wirfs-Brock and Douglas Crockford work session. ecmascript.org wiki. https://www.ecma-international.org/archive/ecmascript/2007/misc/8-16-07_meeting_notes.pdf

W^{3}Techs. 2010. Usage of javascript libraries for websites. W^{3}Techs Web Technology Surveys. April 2010. 2010 data NON-ARCHIVAL https://w3techs.com/technologies/overview/javascript_library/all (SUPERSEDED; also at Internet Archive 23 April 2010 04:16:17). Current data NON-ARCHIVAL https://w3techs.com/technologies/overview/javascript_library/all

Yahoo! Developer Network. 2008. YUI Core. Online documentation. Sept. 2008. NON-ARCHIVAL http://developer.yahoo.com:80/yui/3/yui/#core (BROKEN; also at Internet Archive 10 Sept. 2008 00:24:03).

Alon Zakai. 2011. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (Portland, Oregon, USA) (*OOPSLA '11*). ACM, New York, NY, USA, 301–312. 978-1-4503-0942-4 https://doi.org/10.1145/2048147.2048224

Jamie Zawinski. 1999. the netscape dorm. Web page on www.jwz.org. 8 Nov. 1999. NON-ARCHIVAL https://www.jwz.org/gruntle/nscpdorm.html (SUPERSEDED; also at Internet Archive 8 Nov. 1999 22:25:35). In 2014 the original Web page was changed to begin with a disavowal of the startup culture descibed in the essay. That version is what is currently accessed via NON-ARCHIVAL https://www.jwz.org/gruntle/nscpdorm.html.

Boris Zbarsky. 2014. RE: @@new. es-discuss mailing list. 17 June 2014. NON-ARCHIVAL https://mail.mozilla.org/pipermail/es-discuss/2014-June/037849.html (also at Internet Archive 23 July 2014 23:17:30).