JavaScript: The Used Parts

by

Sharath Chowdary Gude

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 2, 2014

Keywords: JavaScript, Empirical Study, Programming languages

Copyright 2014 by Sharath Chowdary Gude

Approved by

Dr. Munawar Hafiz, Computer Science and Software Engineering
Dr. Levent Yilmaz, Computer Science and Software Engineering
Dr. Jeff Overbey,Computer Science and Software Engineering

Abstract

JavaScript is designed as a scripting language which gained mainstream adoption even without creation of proper formal standard. The success of JavaScript could be attributed to the explosion of the internet and simplicity of its usage. Even though web is still its strongest domain, JavaScript is a language constantly evolving. The language is now used in OS free Desktop application development, databases etc. The standards committee wants to revamp the standard with addition of new features and provide solutions to controversial features of language[1]. This needs a large scale empirical studies spanning all diverse paradigms of JavaScript.

The interpreted nature of language needs a proper mechanism to perform both static and dynamic analysis. These analyses should be analyzed and interpreted to understand the general usage of language by the programming community and formulate the best ways to evolve the language. The inherent misconceptions among the programming community about the language are to be studied, usage patterns are to be analyzed to generate data on usage of features and thus justifying an argument for need to refine the feature.

The thesis explores the features in the language deemed problematic by experts, the way these features are used by the programming community and the need for refining these features backed up with an empirical study. The corpus for our empirical study is larger than any study on JavaScript until now with over million scripts and from variegated sources. The current goal of standards committee is to divorce the language from the perception of being a simple scripting language and get the language evolved to be a mainstream programming language. Our work will help in formulating a new direction for the future standards, justifies the need for proposed changes in next specification ECMAScript 6 and root out widespread myths in programming community.

Acknowledgments

I would like acknowledge all the people who have directly or indirectly helped me through my research.

I am immensely indebted to Dr.Munawar Hafiz, who has been an ideal advisor in every regard. He made me a part of his research team even though he didn't knew a lot about men and has been very supportive ever since. I thank him for introducing me to the joys and frustrations of scientific research and for teaching me ways of self-discipline in scientific research and academic writing.

I would like extend my deepest appreciation for my committee members:Dr.Levent Yilmaz and Dr.Jeff Overbey. for inspiring me with their work. They have been very supportive of my work and were patient with me.

I owe my position to my parents Padma and Venkateswarlu. Finally, I would like to thank my friends in Auburn without whom life wouldnt have been easy.

Table of Contents

Abstract ii

Acknowledgments iii

List of Figures vii

List of Tables viii

1 Introduction 1

1.1 History of JavaScript 1

1.2 The Problem 2

1.3 Thesis statement 4

1.3.1 Corpus 5

1.3.2 Methodology 8

1.3.3 Results 9

1.4 Organization of Thesis 10

2 Strict Mode 12

2.1 Motivation 12

2.2 Approach 13

2.3 Results 14

2.4 Discussion 14

2.4.1 Why is strict mode Unpopular? 14

2.4.2 Proper Usage of Strict mode 16

2.4.3 Concatenation bug 16

2.5 Impact 17

3 With statement 19

3.1 Motivation 19

3.2 Approach 20
3.3 Results 20
3.4 Discussion 21
3.5 Impact 22
4 For..in 23
4.1 Motivation 23
4.2 Approach 23
4.3 Results 24
4.4 Discussion 25
4.5 Impact 26
5 Variable Scope 27
5.1 Introduction 27
5.1.1 Approach 28
5.1.2 Results 29
5.2 Discussion 29
5.3 const keyword 31
5.4 let 32
5.5 Impact 33
6 Function Inside Block 34
6.1 Motivation 34
6.2 Approach 35
6.2.1 Results 35
6.3 Discussion 36
6.4 Impact 37
7 Objects in JavaScript 39
7.1 Motivation 39
7.1.1 Approach 39

7.1.2 Results 39
7.2 Discussion 40
7.3 Impact 41
8 Related Work 43
8.1 Introduction 43
9 Conclusion and Future Work 45

List of Figures

1.1 Repository 4

1.2 Research Corpus 6

1.3 Instrumentation:Collection and Analysis of Scripts 9

2.1 Usage of Strict mode 14

3.1 Usage of 'With' 21

4.1 Usage of 'for..in' 25

5.1 Variable usage in JavaScript 29

5.2 Trends in Variable declaration 30

5.3 const usage in JavaScript 31

6.1 Inner Functions: Statistics 36

6.2 Number of functions inside block 37

7.1 Usage of Native Objects 40

List of Tables

Chapter 1

Introduction

1.1 History of JavaScript

JavaScript was developed by Brendan Eich as a client-side scripting language in 1995 at Netscape. The intention of Netscape was to develop an interpreted language on par with Java that would appeal to the web authors and non professional programmers[2]. JavaScript gained success and was adopted by Microsoft for its browser, Internet Explorer, in 1996. JavaScript is now a trademark owned by Oracle. The widespread adoption of the language has necessitated the need for standardization. The standardized version is named as ECMAScript(ES). Microsoft's implementation of ECMAScript is referred to as JScript. The different dialects of ECMAScript implementation started out because of the trademark issues.

The first edition of ECMAScript was published in 1997 as ECMA-262 specification. The standards process continued with the release of ECMAScript 2 in 1998 and ECMAScript 3 in 1999. No major changes were proposed in ECMAScript 2. ECMAScript 3 provided support for regular expressions, string handling, newer control statements, tighter error definition etc. The ECMAScript 4 has been scrapped prior to release because of the differences in ideology regarding the increasing complexity of the language and version ECMAScript 3.1 is released. ECMAScript 5 is released in 2011. It adds new features including strict mode which disallows error prone features of the language, getters and setters, support for JSON etc. The next version is being planned to be released this year and is called 'ECMAScript 6 Harmony'[3].

JavaScript was initially used by web authors for client-side scripting. The introduction of Ajax has diversified its usage. Now, it is used by major software giants. It is used to

develop Rich Internet Applications (RIA) whose functionality is similar to traditional desktop applications. A wide variety of frameworks and libraries were written to support these efforts . The reach of the language has been ever increasing since then. It has under gone a few paradigm shifts in its usage which includes: server-side programming like Node.js, Rhino, Helma, JavaScript on Rails, etc., OS independent desktop programming (GNOME), NoSQL databases like Apache CouchDB, JavaScript interpreters are also used in Adobe photoshop, Acrobat and Yahoo's widget engine.

Enterprise applications are now written in Node.js framework which use event-driven paradigm and is shown to have better response rate and number of requests handled[4]. The commonJS[5] project was founded to create common standard library for JavaScript other than browsers.

In order to do that, ECMAScript has proposed the following goals to reform the language[6]:

To be a better language for writing complex applications, libraries (possibly including the DOM) shared by those applications and code generators targeting the new edition. To switch to a testable specification, ideally a definitional interpreter hosted mostly in ES5. To improve interoperation, adopting de facto standards where possible, keep versioning as simple and linear as possible. To support a statically verifiable, object-capability secure subset.

1.2 The Problem

JavaScript is a language riddled in misconceptions and semantic confusions. It can be traced back to the language creation. It was initially named LiveScript, but was renamed as JavaScript owing to the tremendous success of Java. This has started a widespread assumption among people from other programming communities that JavaScript is in some way related to Java[2]. But, the language has little in common with Java. The language supports both object-oriented and functional style programming. The language employs

prototype-based inheritance and objects are used in different context compared to other major programming languages.

JavaScript has been the major component of all the web-pages. The success of JavaScript in web development is undeniable. There are currently an approximate of 5 billion web-pages [7]. JavaScript is the language with programming community that has developers with wide variety of expertise.

As Donald Knuth[8] pointed out, designers of the languages and compilers rarely have any idea how general programmers use the programming languages. Designers' notions about language usage by programmers is generally different from the ways programs are being written. JavaScript programmers have a history of using hard-to-optimize features given its simplicity and dynamic nature[1][9].

The domain of JavaScript now has expanded significantly with its usage prevalent in server-side computing, independent Desktop OS applications, databases etc. This requires addition of new features in the language to cater the widening horizons. Also, there are some inconsistent designs inherent in the language. This necessitates evolution of the language and standards committee is planning to revamp the language, weed out deep-rooted misconceptions and inherent bad design choices[10]. This requires empirical studies to understand the usage of features by general programming community. But, conducting an empirical study on a diverse language like JavaScript is difficult.

The problem discussed in the paper is *There have been quite a few empirical studies in the past both on dynamic features and security aspects. The scope of these studies is limited. They have considered only top 100 sites and a suite of benchmarks. These studies are ultimately not essential in the evolution of the language because:*

1. *The sources studied were not reflective of the significant paradigm shifts the language has undergone. A corpus with sources inclusive of different domains in which JavaScript is used has to be created.*

Figure 1.1: Repository

2. *The empirical data may be difficult to analyze and interpret. Usage of a specific feature in one domain may be significantly different from other both in functionality and frequency. These discrepancies are to be observed and the implications of these inconsistencies are to be reported.*

3. *Several features of JavaScript require a simple lexical analysis to sophisticated dynamic analysis, sometimes both in tandem.*

1.3 Thesis statement

Programmers of JavaScript often use JavaScript in confusing and non-standard ways. A widespread empirical study to understand the usage of JavaScript language features (and the corresponding misuses) must account for the diverse ways JavaScript is used and a combination of crude lexical analysis to sophisticated dynamic analysis.

Our empirical study is conducted on a massive corpus which is both diverse and comprehensive reflective of wide variety of paradigms JavaScript supports. The following subsections discuss the rationale in selecting the elements of corpus, methodology employed for analysis and gives a summary of few findings. The corpus has more than a million scripts from five

different sources. There have been a few empirical studies to understand JavaScripts dynamic behavior [11],[12] and its security issues [13][14][15]. These studies have only focused on top 100 websites and suite of benchmark programs. Our corpus is made decidedly diverse with scripts from variegated sources which helps us better understand usage of language by people with different levels of expertise.

1. Top 100 websites(according to Alexa)

2. 70000 random web pages from wild

3. Top 50 mozilla-Addons which are very agile and adopt new features of language readily,

4. The top 3 widely used libraries namely jQuery, YUI and prototype.

5. Node.js programs

Our research analyses ranges from simple lexical analysis to complex dynamic and static analysis of the features. The features are selected with level of need for reform and with assistance from Douglas Crockford's JavaScript:The GoodParts[1]. This empirical study can be extended with ease in future.

1.3.1 Corpus

The existing studies of JavaScript focus on scripts collected from the top web-pages identified by Alexa, well-known JavaScript libraries, and/or benchmark programs. Recent work on existing JavaScript benchmarks suggest that the benchmarks are not representative of real world JavaScript code[11][16]. Alexa pages are a popular source, but even they represent a small and perhaps atypical sample of scripts. Additionally, some studies are on a small corpus, typically on scripts collected from the top 100 Alexa pages.

We collect scripts from various sources so that they represent JavaScript written by people with different levels of expertise and scripts that have gone through different levels of validation process. We want to ensure that the corpus represent the following population:

Script Source	Source URLs/ Programs	# of Scripts	# of Unique Scripts	% Unique	Program Size (KLOC)
Spidered Pages	66,145	3,621,184	1,041,325	28.76	68,737
Alexa Pages	100	421,317	124,828	29.63	10,522
Firefox Addons	50	1,074	991	92.27	164
JS Libraries	3	3	3	100.00	7
Node.js Appl.	50	2,653	2,548	96.04	1,107
		4,046,231	1,169,695		80,537

Figure 1.2: Research Corpus

1. Scripts collected from the wild written by developers of various skill levels.
2. Scripts written for popular web-pages that are perhaps written more carefully by experienced developers.
3. Scripts that have gone through a validation process and contain newly introduced JavaScript features.
4. Scripts written by experts and widely reused as libraries.

Scripts collected from the wild. Using a web crawler (i.e Win Web Crawler), we initially collected 80,000 URLs. Duplicates among the URL list are removed and we ended with 66,145 URLs. We wrote a perl script that creates an instance of instrumented browser(Mozilla Firefox v21.0a) and loads an URL from list. The script destroys and relaunches the instance of browser loading the subsequent URL in the list after every 10 seconds.

Many of the scripts were not unique; they reused some JavaScript libraries (e.g., jQuery) or other scripts. We used the MD5 hash of a script to identify scripts that were syntactically similar. After removing these duplicates, there were 1,041,325 unique scripts. We refer to these scripts as Spidered Pages in the rest of the document. The scripts were collected in November 2013.

Scripts written from popular pages. We browsed the top 100 Alexa websites as they should contain scripts written by experienced developers. We used Win Web Crawler to automatically crawl each of the top 100 sites and collect URLs; then we loaded the URLs in the instrumented browser. Some banking websites, such as Bank of America, Chase, etc., require a back account to delve into the site. We excluded these sites and proceeded to the next top sites. In total, we have extracted 124,828 unique scripts. We refer to this source as Alexa Pages. The scripts were collected in November 2013.

Scripts that have been validated. We extracted scripts from the top 50 Firefox Add-ons (25 most popular and 25 highest rated) during August 2012. These scripts enhance user experience (e.g., Greasemonkey, Download Statusbar, etc.), enable safe and secure browsing (e.g., Adblock Plus, NoScript, etc.), and provide power tools for developers (e.g., Firebug, etc.) Since Mozilla uses volunteers to vet each new extension and revision before posting it on their official list of approved Firefox Add-ons, these scripts had gone through some validation process. Additionally, some of these scripts contain new ECMAScript features (Mozilla Firefox actively adopts proposed new features). In total, we collected 1,074 scripts from Add-ons, out of which there were 991 (92.27%) unique pages; we refer to this source as Firefox Add-ons in the rest of the paper.

Scripts used as libraries. We surveyed 3 popular libraries: Prototype JavaScript framework version 1.7, jQuery version 1.8.3 and YUI version 3.6.0. We used the minified versions in our study as they are most commonly used libraries. We refer to this source as JavaScript Libraries in the rest of the paper. Figure 1.3 describes a summary of the scripts collected from various sources.

Scripts from Node.js Applications. We collected scripts from Top 50 applications from npmjs.org. Node applications uses Google's V8 engine as its interpreter. We instrumented the V8 interpreter to analyze the dynamic features of the language. Since, these applications are written in object-oriented style, we studied OO features of the language namely mixins and object creation.

In total, our survey is based on over million unique scripts. When we statically counted a feature in our study, we used the unique scripts. How ever, in order to study the dynamic properties of the scripts, we had to load them in an instrumented browser, therefore running all the scripts to collect results.

Email based survey on JavaScript developers In order to understand the reasons behind some of the choices made by developers, we launched an empirical study on JavaScript developers. We concentrated on developers of Node.js applications and in this study. They provided the perspective of the new brand of JavaScript developers.

We selected the participants from the developers of top 100 most starred Node.js applications as listed in www.npmjs. org. We also considered the top 100 most prolific developers listed in the same site. From these sources, there were 137 developers. We sent them an email with seven questions. Three questions asked about their design choices: whether they used strict mode, whether they declare variables in block scope, and whether they create custom objects in their applications. The remaining four were open-ended questions asked about the reasons behind their choicesone each for the three questions, and one about their use of mixins to extend objects. We sent the email with the questionnaire only once. We received 45 responses ($45/137 \approx 32.86\%$ response rate).

More details about the corpus are available on the project webpage:
http://www.munawarhafiz.com/research/jssurvey.

1.3.2 Methodology

The idea behind the empirical study is to identify the usage of particular features by the programmer. Spidermonkey, the JavaScript engine in the Mozilla-Firefox browser has been instrumented to print out the JavaScript from each website opened along with the generation the log reports of the feature encountered and frequency of the usage of the feature. The first 4 components are analyzed by using the instrumented Firefox browser. The features are analyzed by 3 different methods:

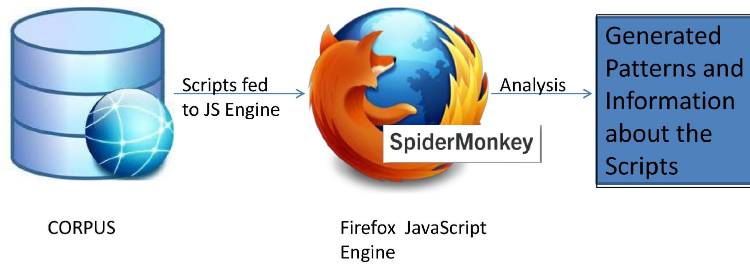

Figure 1.3: Instrumentation:Collection and Analysis of Scripts

1. lexical count
2. parser instrumentation
3. interpreter instrumentation.

The entire process is automated by passing list of URLs to a perl script which sequentially opens each URL in the instrumented browser after a pre-defined time, thereby periodically destroying and invoking the new instance of instrumented-browser. A different methodology is used to analyze each member in corpus.

For top 100 websites, a spider is used to generate 80 URLs from each website and concatenate into a single file which is passed to aforementioned perl script. The random list of URLs from wild is obtained by spidering the web using a keyword and we've used WinWebCrawler[17] for our research.

Node.js applications use an embedded JavaScript engine, Google's V8 for interpretation of JavaScript. We've instrumented the the JavaScript engine to trace the features and generate the log report. We've ran 50 Node.js applications and generated trace files and log reports. The 50 Node.js applications are selected based upon their popularity and permissive licenses.

1.3.3 Results

We discovered several important usage facts that impact how JavaScript can evolve and how its usage patterns can assist developers and tool builders.

- We found confusion and misconception about recently introduced features that impact their adoption (strict mode, Section 2). This suggests that future extensions to the language will need to be carefully and broadly introduced to JavaScript web developers in order for such features to have any significant impact.
- We found continuing misuse of existing problematic features including block level declarations (Section 5) and for...in iteration statements (Section 4). This validates the inclusion of improved alternatives that have been proposed for the next ECMAScript revision.
- We found that adoption of non-standardized and deprecated features make it difficult to introduce new features (function in block, Section 6) and correct past mistakes (with, Section 3).
- We found wide use of functional programming constructs but virtually no examples of scripts defining new object abstraction (Section 7). This last result is surprising and suggests the need for further studies to see whether this reflects a language deficiency or is just a legacy of the simplistic way JavaScript was used on web pages during the first decade of its existence.
- We found that overwhelming majority of Node.js developers do infact use mixins. They employ variety of strategies to achieve that.

1.4 Organization of Thesis

This study focuses on collecting usage data of recently introduced features and features that will be modified/deprecated in the next major ECMAScript revision. Specifically, we analyze usage questions concerning ECMAScript 5 strict mode; usage of the with statement; the use of block level variable declarations; the use of functions defined in blocks;

object property enumerations using the **for...in** statement; and the definition of new object abstractions. Members of standards committee deemed these features to be among the problematic and are needed to be addressed in the future revisions of the language.

Strict mode is a recently introduced language feature. The **with** statement is a legacy statement deprecated in strict mode. Block level scoping, although not currently supported, is being considered as a new feature in the next revision. Usage patterns of the **for...in** statement that enumerates over object properties provides clues about how JavaScript programmers currently make use of objects and inheritance and are relevant to the design of new enumeration mechanism in the next ECMAScript revision. Analysis of object abstraction patterns provide information about how often JavaScript programmers define new classes of objects. Sections 2 to 9 provide the details.

The thesis has been organized into different chapters with each feature discussed in a specific chapter. In chapter 9, we discuss our conclusions and future work.

Chapter 2

Strict Mode

2.1 Motivation

Strict mode was introduced in ECMAScript5. The intent of introducing strict mode is to gradually eliminate the error-prone and hard-to-optimize features in the future revisions of the ECMAScript.

Strict mode allows the users to switch to a more restrictive version of JavaScript which disallows usage of certain features (e.g with).

The major changes in strict mode[18] compared to the normal semantics are:

1. throw errors instead of the silent errors in previous versions of standard
2. disallow the usage of features that are bound to make script optimizations slower and pose security issues. It includes disallowing new scope in eval, aliasing properties of arguments objects created within it, prevents the exposure of global variables to outside scripts.
3. prohibits use of the syntax and keywords which are saved for future versions of the language.

Strict mode is designed to be backward compatible with browsers that do not support it. Even in absence of strict mode support in the browsers, no side effects are observed. It will be treated as a string literal that isn't further used in the script.

Strict mode can be enforced on the script by using the directive *use strict*.

The directive can be used in two ways: 1) declaring *use strict* at script level as a declarative prologue thus enforcing strict mode on the entire content of that file, or 2)

Listing 2.1: file-level declaration

```
"use strict";
function testStrictMode(){
 var newVar=2;
    return newVar;
  }
```

Listing 2.2: function-level declaration

```
function testStrictMode(){
 "use strict";
  var newVar=2;
    return newVar;
  }
```

declaring *use strict* within the function body thus enforcing strict mode only on the function and any nested functions. Script-level declaration may cause a concatenation bug when a script written using strict mode is concatenated with third- party scripts that depend upon legacy features or semantics that are unavailable in strict mode.

Our research focuses on the level of adoption of the feature and explores the inconsistencies and problems faced by the programmers in the usage of the feature.

The code snippets in listing 2.1 and listing 2.2 show file level and function level declarations respectively.

2.2 Approach

The scripts are lexically searched to detect the presence of the directive 'use strict' and are separated. Regular Expressions are employed to categorize the usage of directive into file level and function level declarations. We also surveyed developers to explore the reasons why they use strict mode,if not, why they refrain from using it.

Script Source	# Unique Scripts Using `use strict`	% Unique Scripts Using `use strict`	`use strict` instances	
			File Level	Fn Level
Spidered Pages	9,265	0.89	239	27,384
Alexa Pages	965	0.77	93	2,941
Firefox Addons	85	6.50	73	12
JS Libraries	0	0	0	0
Node.js Appl.	237	9.30	166	135

Figure 2.1: Usage of Strict mode

2.3 Results

The results are tabulated in figure 2.1.

The level of adoption of strict mode is low among all the elements of corpus. The percentage of adoption is still lower than 1 percent among all components of corpus. It can be also observed that the function level declaration is disproportionately higher than file level declaration(i.e. declarative prologue) with them being 27384, 239 and 2941, 93 in Spidered pages and Alexa pages respectively. JSLibraries have shown no use of strict mode. This could be attributed to possibility of concatenation bug discussed in the subsection below. Only 19 out of 100 Alexa sites used Strict mode. Of the 50 Add-ons in our corpus, only 5 use strict mode. Most programmers declare use strict at function level following the recommended practice. But the trend is opposite in Add-ons. Developers of Add-ons prefer using use strict as a declarative prologue to the entire script.

2.4 Discussion

2.4.1 Why is strict mode Unpopular?

The lack of adoption of strict mode can be attributed to 3 major misconceptions among developers:

1. Developers cannot use features such as strict mode unless it is supported by all the widely used web browsers. Strict mode is not fully supported by all browsers (i.e Opera mini). Also, We have to factor in lot of end users who haven't yet updated to latest versions of browsers. Microsoft Internet Explorer did not support strict mode until IE10 (October 2012), even though strict mode has been a part of ECMAScript 5 since December 2009.

2. There is a common notion that implementation of strict mode is different in different browsers. Usage of Arguments.callee in strict mode raises TypeError in Opera, Firefox and is valid in Chrome. Developers refrain from using it to make their code consistent across browsers.

3. Another popular belief is that strict mode should only be used during development for debugging, but not in production code. They believe that the third-party tools (JSHint/JSLint) also provide similar benefits as strict mode. Thus, strict mode may be used by developers but it is not shown in our corpus.

These misconceptions were observed in the e-mail based survey on developers of Node.js applications and Mozilla Add-on developers. 17 out of 45 developers($\approx 37\%$) have used strict mode in their code. The responses justifying the choice of refraining from using strict mode are mostly misconceptions.

Compatibility issues because of lack of support is one major misconception. 10 developers ($10/45 \approx 22.22\%$) were concerned that strict mode will change the semantics of existing code or will break it.

The lack of browser support will treat the directive more like a comment. One change in semantics in strict mode is in the scoping of declarations contained within direct eval calls. We searched the entire corpus and did not find any instance where eval is used in strict mode. However, this may be because programmers removed eval from their code when introducing strict mode which is recommended [19]. The other concern about mixing strict

and non-strict code is also a misconception. Strict mode is lexically scoped to a script or function body; it has no global effect. Concatenation has problematic issues, but using separate strict and non-strict scripts within the same program is just fine.

A surprisingly high number of developers, even two of those who admit using strict mode, mentioned that strict mode is unnecessary (14, 14/45 $\approx$ 31.11%). 4 developers (4/45 $\approx$ 8.89%) mentioned that the same benefits can be found from JSLint and JSHint. A few others (4, 4/45 $\approx$ 8.89%) think that the mode is too restrictive, and do not support their favorite features (Lack of octal mode is my only complaint).

2.4.2 Proper Usage of Strict mode

The results show us that programmers of both Spidered pages and Alexa top sites prefer function level strict mode compared to file level declarations as recommended. File-level declarations are fine as long as the scripts are not blindly concatenated thereby avoiding concatenation bugs. Contrary to other parts of corpus, programmers of Firefox Add-ons prefer file level strict mode declaration as the scripts are typically independent and possibility of concatenation bugs is nullified.

JavaScript libraries do not use strict mode. Library writers may believe there are issues with using strict mode but that is basically the same misunderstanding about the nature of strict mode. It is only when library code is physically embedded (discussed next section) that there is a possible problem.

2.4.3 Concatenation bug

Blind concatenation of scripts is shown to have caused concatenation bug. It's a common practice to concatenate the third-party scripts with the developers code and any file level declaration of *use strict* in the third party script makes both scripts strict. It thus disallows features in the developers code as it is intended to be non-strict code thus breaking it.

Listing 2.3: Concatenation Bug

```
<<JSON2.js code inserted here>>
...
multi_price_block: function(deal) {
var self = this; var priceHash = self.get_prices(deal);
...
if ( !priceHash[ ourPrice ].isRange ) {
ourPercentOff = Math.round
((listPriceAmount - ourPriceAmount)* 100 / listPriceAmount);
...}
...};
```

We countered few concatenation bugs(less than 10) while running instrumented browser to load the list of URLs. One of the major website, Amazon has also suffered from a concatenation bug. The 'lightning deals' page during a major holiday caused the bug. JSON2.js, the third party script which uses file level strict mode declaration is blindly concatenated to the Amazon's script enforcing strict mode on Amazon's code which are meant to be non-strict.

The variable ourPercentOff is not explicitly declared. The Amazon's script written in non strict mode makes it a global variable. The strict mode enforced on it throws an error as the variable is not explicitly declared.

2.5 Impact

Language designers should be concerned about strict mode and similar confusions about newly-introduced features. A possible reason for this misconception is that JavaScript developers do not receive information from a single source. Unlike single vendor languages, there is not a coordinated launch when a new version of JavaScript becomes available. Instead, it rolls out at different times in different browsers sometimes in a piecemeal fashion.

Some developers complained about strict mode not supporting their favorite features. Such concerns should be addressed during language evolution. For example, ES6 introduces a new syntax of octal literals: 0o777.

Chapter 3

With statement

3.1 Motivation

With keyword is the widely debated to be harmful feature in the language, deemed 'awful' by crockford [20]. It was introduced to provide short hand notation to refer object properties to improve the readability. The usage of feature makes compiler optimization harder and reduces the performance of script with constant scope resolution at runtime and is known to cause missed recursion. The missed recursion causes global namespace pollution. ECMAScript standards committee is in favor of abandoning the feature given the disadvantages it offers and complete replaceability of the feature with simple idioms. A simple reference to variable can be created as shown in table to provide the shorthand notation that **with** does. Several such idioms can be used to replace **with**.

Despite the suggestions from the experts, the usage of feature is still prevalent. Our research focuses on the level of prevalence of the feature, the proportion of property modifiers within the **with** that potentially cause global namespace pollution.

Listing 3.1: sample code

```
var x = document.body.scrollLeft;
document.write('text1');
document.write('text2');
document.write('text3');
```

Listing 3.2: with usage to improve readability

```
with document {
var x = body.scrollLeft;
write('text1');
write('text2');
write('text3');
}
```

Listing 3.3: With Replaceability

```
var d = document;
var x = d.body.scrollLeft;
d.write('text1');
d.write('text2');
d.write('text3');
```

3.2 Approach

We counted the number of **with** statements in unique scripts by searching lexically. In order to detect the dynamic behavior of **with** statement, we instrumented the interpreter component (jsinterp.cpp) of Firefox: 14 lines of code were added. Scripts from Firefox Add-ons pages and JavaScript libraries were analyzed manually. This is because a client that exercises a library (or an Add-on) may not execute all parts of the code.

3.3 Results

With is a dynamic feature where scope of properties used are resolved at runtime. Our research analyzes the number of **with** keywords within the corpus, the percentage of properties resolved in the **with** scope and percentage of those that have escaped to global space.

Results shows limited but significant presence of **with** usage given the controversy it generated over years. Less than 1% scripts use the **with** statement. Most of the properties are resolved within the the scope created by **with**. Another interesting observation is vast majority of scripts containing **with** are from two sites Amazon and Ebay. Together, they

Script Source	# Unique Scripts Using **with**	% Unique Scripts Using **with**	# of **with** instances	# properties resolved in global	% properties resolved in global
Spidered Pages	6,237	0.60	8,645	1,589	11.71
Alexa Pages	718	0.58	1,193	719	10.82
Firefox Addons	87	8.77	87	0	0
JS Libraries	0	0	0	0	0

Figure 3.1: Usage of 'With'

contribute upto 60% of scripts to Alexa. Percentage of scripts using **with** in spidered pages and alexa pages is 0.69 and 0.58 respectively. The total number of properties used within the **with** are 13419, 6332 in Spidered and Alexa pages. Of these, 11834 and 5613 properties are resolved within the scope of **with** among the Spidered and Alexa pages. The rest of properties are resolved in the global scope(11.71% in spidered pages, 10.82% in Alexa pages.)

The JavaScript libraries employed in our research have no instances of **with**. Among the 50 Mozilla Firefox Add-ons in our corpus, 11 have used **with**. Lack of DOM and scripts being stand-alone cannot pollute global name space in Add-ons. Two Add-ons have accounted for most usage of **with-s**:NamFox, an automated marker for NeoSeeker has 34 and Feedly, a Google reader Add-on has 25.

3.4 Discussion

Most of the **with** statements are resolved within the object. The global objects on which **with** is used are predominantly objects for DOM access and window objects. For example, we found 627 properties resolved in the global CSSProperties object, and 403 in window object in spidered pages.

Prior to the introduction of 'let' keyword, **with** statement is used to mimic block scope in JavaScript, but at the expense of performance.

3.5 Impact

ECMAScript 5 strict mode disallows **with** since it introduces dynamic scope. It is unlikely that **with** statement will ever be eliminated, because that would break existing web pages. However, the committee hopes that use of strict mode will minimize the usage of **with** in newly created content.

Our results also show that only a few instances of **with** are resolved in global space. The remainder can be replaced with simple idioms According to park at al.[14], rewritability of every pattern of **with** is possible except for generator pattern because of the function which generates dynamic code within the statement. This usage pattern account for less than 1% usage in the real world. This provides an ample opportunity to develop refactoring tools to replace **with** improving performance and maintainability when the code is used with third party strict scripts.

Chapter 4

For..in

4.1 Motivation

for..in construct is the cross-browser technique used to iterate through the properties of a generic object. This construct not only iterates through the non-shadowed, user defined properties of the current object but also the properties inherited from the prototype chain. The **hasOwnProperty** prevents the enumeration over the properties inherited from the prototype chain. Addition of any property to object prototype(i.e Object.prototype) with in the loop results in a potential error because of the iteration of inherited property. Another commonly found misconception associated with **for..in** is its use for array Enumeration. This has been ill-advised as the order of iteration of elements is not guaranteed with the **for..in** loop and the inherited properties are also iterated when used with **for..in** construct. Also, these practices eliminates the possibility to extend the pre-defined objects like Array, Object etc.

Our research aims to generate the statistics on the level of usage of **hasOwnProperty** in tandem with **for..in** and usage of array enumeration using **for..in**. A survey on developers is done to shine light on their reason to not use **hasOwnProperty** with **for..in**

4.2 Approach

We instrumented the parser component of Mozilla Firefox to identify the **for...in** statements and corresponding **hasOwnProperty** uses. For every for...in statement, we traced

Listing 4.1: problem with for..in usage for Array Enumeration

```
Array.prototype.foo = "foo";
var array = ['1', '2', '3'];

for (var i in array) {
  alert(array[i]);
}
// alerts 1,2,3,foo
```

Listing 4.2: Use of hasOwnProperty with for..in

```
Object.prototype.foo = "foo";
var person={fname:"John",lname:"Doe",age:25};
for (var x in person)
  {
  if(person.hasOwnProperty(x))
  alert(x);
  }
  //alerts John,Doe,25
  //alerts John,Doe,25,foo without hasOwnProperty()
```

whether it is used with **hasOwnProperty**. Finding whether a **for...in** is used for array enumeration is similar except it is handled by a separate portion of the code inside the parser component. Regular expressions are used to generate the statistics of Firefox Add-ons.

4.3 Results

The results table below shows the **for..in** alongside **hasOwnProperty**. **for..in** construct in Alexa pages are better filtered with **hasOwnProperty**. Of the 126928, 11344 **for..ins** encountered in Spidered and Alexa pages, 19645, 4138 of them are filtered with **hasOwnProperty** construct. The usage of **hasOwnproperty** is doubled in Alexa pages compared to Spidered pages with the usage percentage being 36.4 and 15.8 in Alexa and Spidered pages respectively. Surprisingly, the JSLibraries and Mozilla Add-ons are at the exact opposite end of spectrum. **hasOwnProperty** is rarely used to filter **for..in** in Firefox Add-ons(only 6.6%)

Script Source	Number of Unique Scripts using 'for..in'	Percentage of Scripts using 'for..in'	No. of 'for..in' instances in Unique Scripts	Percent of for..in filtered with hasOwnProperty	Percentage of for.ins used for Array Enumeration
Spidered Pages	72341	7.51 %	126928	15.8 %	2.41 %
Alexa Pages	6013	5.02 %	11344	36.4 %	1.8 %
Firefox Addons	295	29.7 %	384	6.65 %	0.92 %
JSLibraries	3	100 %	92	61.8 %	0 %

Figure 4.1: Usage of 'for..in'

Table 4.1 also aggregates the instances of **for...in** that are used to enumerate through arrays. Developers of JavaScript libraries do not use **for...in** this way. Of the 126928, 11344 **for..ins** encountered in Spidered and Alexa pages, only 2332(3.21%), 243(3.93%) are used for Array enumeration.

4.4 Discussion

The results inform us that most of the **for..in-s** are not filtered with **hasOwnProperty**. People do not follow expert's suggestion in this regard. It could be attributed to the fact that most of the objects are stand-alone with only one parent(i.e Object.prototype). This may have prompted the developers to discount the usage of **hasOwnProperty**. But, it does prohibit the developers extend the pre-defined objects.

The another conclusion that can be drawn from results is that developers typically do not favor inheritance. Most objects being stand alone solidifies it. JSLibraries show greater usage of **hasOwnProperty** in correspondence with **for...in** An interesting observation shows that jQuery, a library coded in functional oriented paradigm has no instance of **hasOwnProperty** in tandem with **for..in**. While, YUI, object-oriented library has each of its **for..in** filtered with **hasOwnProperty**.

A common JavaScript pattern is to define all methods as properties of prototype object and all data defined as own properties of instance object that inherit from such prototypes.

A typical use of **for...in** only wants to see data properties. But prior to ECMAScript 5, lack of any efficient way to define method properties as non-enumerable prompted experts recommend **hasOwnProperty**. ECMAScript 5 introduces the defineProperty function that can explicitly set nature of property as either enumerable or non-enumerable The usage of defineProperty is seldom observed in the corpus with 43 instances in all of wild pages. This is probably because of the relatively recent availability of ECMAScript 5 level JavaScript implementations.

It has been ill advised to use **for..in** for array enumeration as not only the elements but properties of Array are iterated. A simple for loop as in other programming language is a better alternative. Another observation regarding **for..in** is that multiple instances of the variable used for iteration not being declared explicitly is identified. The variable becomes global creating global namespace pollution.

4.5 Impact

The high percentage of **for...in** loops that do not contain **hasOwnProperty** checks may indicate that it is uncommon for JavaScript programs to define their object abstractions with shared prototype methods. Instead they may be primarily using the built-in objects abstractions whose methods are not enumerated by **for...in**. Another observation is that definition of new object-based abstractions are seldom found in the corpus.

A **for...of** statement has been proposed for inclusion in the next revision of the ECMAScript standard. **for..of** iterates over property values unlike **for..in** which iterates over property names. Another level of abstract iterator construct is used to iterate over the array elements in the indexed order and will not produce any non-array properties(own or inherited). The stats confirming the use of **for...in** to iterate over arrays suggests ignorance on the part of programmers. It also suggests that there will be a need to educate developers to use **for...of** instead of **for...in** and that there is probably a need for tools to support refactoring of **for...in** statements into **for...of** statements.

Chapter 5

Variable Scope

5.1 Introduction

Variable scope is one of the confusing semantics in JavaScript especially for developers from mainstream languages like Java, C++. The language only has a global scope and a functional scope. It ignores the concept of block scope. Any function defined in the scope creates a nested scope. The scope resolution of the variable starts with inner most scope and works its way outward to global scope. The with statement can be used to create block scoping but it creates a myriad of problems as mentioned in chapter 3.

The code snippet shows the different kind of variable declarations and usage we've worked upon our corpus. There are five usage patterns within function bodies.

1. A variable is declared in function scope and used exclusively in function scope, i.e., not used inside any blocks (functionScopeOnly).

2. A variable is declared in function scope but used exclusively in block scope (blockScopeOnly).

3. A variable is declared in function scope and used in both scopes (bothScopes).

4. A variable is declared in block scope and used exclusively in block scope (blockRestricted).

5. A variable is declared in block scope but used outside the block scope (escapedBlockScope).

Listing 5.1: Scoping in JavaScript

```
var GlobalScope;
function VariableScope() {
var blockScopeOnly = ...;
var functionScopeOnly = ...;
var bothScopes = ...;
if (functionScopeOnly < 20) {
console.log (bothScopes);
var blockRestricted, escapedBlockScope;
blockRestricted = blockScopeOnly;
}
escapedBlockScope = ...;
}
```

6. A variable declared in global scope(GlobalScope)

Crockford[1] suggests that variables are to be hoisted to the functional scope to prevent the confusion of lack of block scope in JavaScript. we tried to answer the following research questions: Do programmers actually practice the variable hoisting? if they didn't, do they respect the block scope and use the variables declared inside the block within the block? The answers do support the introduction of proposed construct in upcoming revision of standard, ECMAScript 6.

5.1.1 Approach

We instrumented the parser component of Mozilla Firefox to identify the scope of variables. We automatically loaded the scripts from spidered pages, JSLibraries and Alexa pages in the instrumented browser. The instrumentation keeps track of all variables declarations and updates their scopes every time a new use of a variable is encountered; this is done per function. For scripts in Firefox Add-ons, we manually counted all variables and their scopes. We also asked developers about whether they declare variables inside blocks or hoist variable declarations at function level; we asked them to justify their choices.

Script Source	Declared at Function Level			Declared within Block		Total No of Variables
	% Function Scope Only	% Block Scope Only	% Both Scopes	% Block Scope Only	% Escaped Block Scope	
Spidered Pages	0.25	0.04	71.64	25.68	2.39	24,228,649
Alexa Pages	1.03	0.43	69.88	26.93	1.73	864,622
Firefox Addons	8.69	3.66	60.12	27.53	0	11,918
JS Libraries	0.38	0.22	85.88	13.15	0.37	4,574

Figure 5.1: Variable usage in JavaScript

5.1.2 Results

The table below show us the statistics of the listed usage patterns. All of the components in corpus have shown a common trend of declaring variables in functional scope and using them in both scopes accounting for nearly 68%. 2.34, 1.74 % of variables declared with in the block among Alexa and spidered pages have been used outside the block which shows us that a limited albeit substantial amount of programmers do not respect the block scope.

5.2 Discussion

The results of the five usages are shown in the figure. It shows that programmers prefer declaring variables in the function scope even if used in block scope. But, there are still significant portion of the variables declared inside block scope. A considerable portion of the block scope declared variables escaped block scope in both Alexa and spidered pages. This coupled with JavaScript's use of anonymous functions and event handlers worsen the understanding of scope. Developers of both Mozilla Firefox Add-ons and JavaScript libraries did try use the variable declared in the block within the block mimicking block scope from other mainstream languages.

The results show that programmers do not necessarily follow Crockfords advice and hoist block-scoped variable declarations to the top of functions. Although there are more than 10,000 instances of variables whose declarations have been hoisted to function scope in

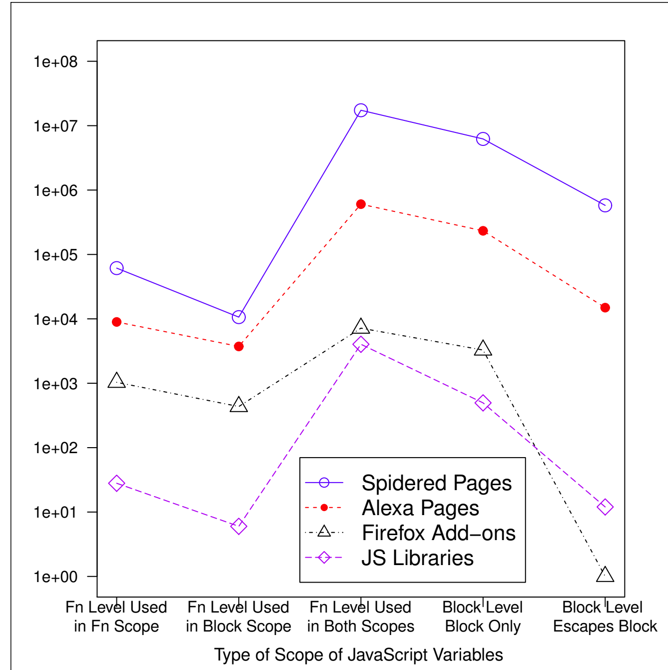

Figure 5.2: Trends in Variable declaration

spidered pages, the proportion is very low (0.04 %) and in fact one order lower than variables declared within block. Another interesting issue is that there are more variables that escape block scope compared to variables that are hoisted.

Survey As mentioned earlier we've conducted a survey on 46 developers. We've posed them the questions regarding their usage of scope in JavaScript and usage of newly introduced *let*. 34 developers have responded to this particular list of questionnaire. 23 developers(67.4 %) have stated that they embraced the functional scope of JavaScript and didn't try to mimic block scope. The usage of linters which block declaration of variables inside blocks, minimization of hoisting-related surprises are the prominent reasons developers embrace this style. 11 developers have stated they tried to imitate the block scope because of the improved readability and maintainability it provides. A couple of responses from our respondents summarizes their justification in usage of such style: *prefer to have the definitions as close to the uses as possible, to prevent action-at-a-distance and refactoring mistakes*, *Principle of Least Privilege[21] and give variables only the smallest amount of scope needed*

Listing 5.2: Const behavior in Chrome and Firefox

```
const beatcop = 'On Patrol?';
alert(beatcop); // On Patrol?

beatcop = 41;  // Fails silently
alert(beatcop); // On Patrol?

var beatcop = 'Sequel?';//TypeError: redeclaration of const beatcop
```

Script Source	Unique Scripts with 'const'	No.of 'const' instances
Spidered Pages	0	0
Alexa Pages	0	0
Firefox Addons	333	1625
JSLibraries	0	0

Figure 5.3: const usage in JavaScript

5.3 const keyword

JavaScript doesn't have a keyword as of yet to define constant variables. A recent keyword **const** has been introduced in ECMAScript 5 to define a constant, global or local to the function. A constant variable cannot be redefined or reinitialized with its scope. **const** variables cannot share the name with other functions or variables within its scope. **Const** variables follow same scoping rules as the variables. The semantics of the **const** keyword are supposed to be changed in next iteration of standard. It has been proposed to create block level scoping like **let** in further iterations of the language. **Const** keyword is currently supported by recent versions of browsers IE, Firefox, Chrome, Opera but reassignment is thrown as TypeError only in Firefox and Chrome.

Results:

Listing 5.3: let behaviour in Chrome and Firefox

```
var x = 10;
if(x>5){
let x=5;
alert(x);//alerts 5
}
alert(x);//alerts 10
```

The usage of **const** in the corpus is detailed in figure below. We've identified 43 and 77 instances of **const** in both Alexa and Spidered Pages. But, most of them are from the internal firefox scripts. Firefox has adopted features in the standard yet to be implemented. Programmers currently use various conventions and tactics to define constants. Some of them include block letter variable names and returning value from function. The introduction of the feature will make it less cumbersome.

5.4 let

let keyword is used to mimic the concept of block scope prevalent in all other programming languages. Thus, it limits the scope of variable to block, statement or expression. It is part of ECMAScript's newly proposed set of features to assimilate the language with other main stream languages. Earlier **with** was used to provide block scope. The keyword **let** provides block scoping without the disadvantages of the former. Redeclaration of a **let** variable within a block or a function would throw TypeError. One major advantage is that we can finally bind variables locally into the scope instead of the functional scope or global scope(with var). **let** keyword is implemented in recent versions of major browsers but hasn't been implemented in server side frameworks like Node.js, Rhino.

Results:

Mozilla supports the **let** construct to declare block-scoped variables. 11 out of 50 Firefox Add-ons from our corpus use **let**. There are 1,800 variable declarations with **let**. In the table, they are listed as block-restricted variables.

The survey also targeted the awareness and usage of let in developer community. 6(17.4%) of the developers said that they have started using *let* for Mozilla Add-ons and Node.js applications(other than browsers). Several developers have shown excitement about the introduction of the feature.

5.5 Impact

JavaScript standards committee wants the language to be the considered as a serious programming language rather than a simple scripting language. The introduction of block scope could reduce the confusion for programmers from other languages.

Developers of Firefox have already included the support for let construct(introduced in ECMAScript 6) in their browser. Several Add-ons did use the **let** keyword to enforce stricter block scope. We have identified more than 1800 instances of variable declarations with **let**.

Because hoisting a variable is sometimes desired and sometimes has to avoided, JavaScript IDE designers can think of adding refactorings to support this activity. Hoisting a variable from block scope to function scope is an obvious refactoring. Another refactoring can redeclare a variable (previously declared with var) inside a block scope using **let**, when it can determine that the variable is used in block scope only. Yet another refactoring can push a variable from function scope to block scope and promote the use of **let**.

Chapter 6

Function Inside Block

6.1 Motivation

JavaScript treats functions as first class objects; they can be created and modified dynamically and passed as data to other functions and objects. JavaScript allows functions to be defined inside other functions unlike other programming languages.

Nesting functions can cause adverse effect on performance if done incorrectly. The creation of inner functions is always deferred i.e. created at run time. The nested function is created each time outer function is called. Thus, reference to inner function has to be created and destroyed repeatedly.

According to ECMAScript standard, blocks are allowed to have statements only[22]. But, various implementations(e.g Firefox, Chrome) does support function declaration within the block against the recommendation of the standard. There are major differences in the behavior of these implementations.

In few implementations, the function declaration with in the block is hoisted making it available outside the block irrespective of conditional statement. Few other implementations treat it as a conditional statement available upon the successful evaluation of the condition. Thus, code cannot be reliably ported across the various implementations/browsers. ECMAScript standard committee is planning to provide alternate means for declaring functions in a statement context[23]. .

The figure 6.1 shows that the behavior of functions declared in the block is varied in various implementations and hence deemed a poor programming practice. We tried to answer following research questions: Do programmers actually define functions in the block against the recommendations? How widespread is the practice? Are the functions defined in the

Listing 6.1: Function in Block:Different implementations

```
var abc = '';
if(1 === 0){
    function a(){
        abc = 7;
    }
}else if('foo' === 'foo'){
    function a(){
        abc = 19;
    }
}else if('foo' === 'bar'){
    function a(){
        abc = 'foo';
    }
}
a();
document.write(abc);
//Mozilla's engine outputs 19.
//chrome's engine ouputs foo
```

block called from outside the block? The planned implementation will break the existing code if they have inner functions that escape block scope.

6.2 Approach

We instrumented the parser component of Mozilla Firefox to count the inner functions. We automatically loaded the scripts from Spidered pages and Alexa pages in the instrumented browser. When the parser encounters a function declaration, it identifies whether the function is declared inside a block; it also identifies the type of block the function is in.

6.2.1 Results

The results are tabulated below. Most of the functions are declared outside the block. Low but a significant proportion of functions are declared in the block in both Alexa and

Script Source	No. of functions declared in function scope	No. of functions declared in block scope	Percentage of inner functions in function scope	Percentage of inner functions in block scope
Spidered Pages	2083659	43538	97.96 %	2.04 %
Alexa Pages	67115	2411	95.4 %	4.6 %

Figure 6.1: Inner Functions: Statistics

Spidered pages. Surprisingly, we've found greater proportion of function declared inside block for Alexa pages. Most of the functions declared inside block are within the blocks if and try/catch. 37 and 236 instances of functions in Alexa and Spidered pages declared inside the block have escaped.

6.3 Discussion

Defining a function within functions is heavily embraced in JavaScript. This could be attributed to the heavy functional oriented programming[1]. Both, anonymous functions and named functions are widely used as inner functions. The results does show that declaration of function inside the block is low albeit significant(43538). ECMAScript wants to address the issue as such declaration was never allowed in the standard. The revisions proposed will cause significant impact on the existing web content and may even break a little.

Similar trends of function usage was observed in Alexa pages as well. Apparently, most of the functions are declared inside an if statement block, and try and catch statement blocks (we did not distinguish the blocks because the parser handles them together).

For example, in spidered pages, among the total 68,851 functions inside blocks, most (40612, 93 percent) are declared within an if blocks, while a only a few (456, 1.06 percent) inside a for block.

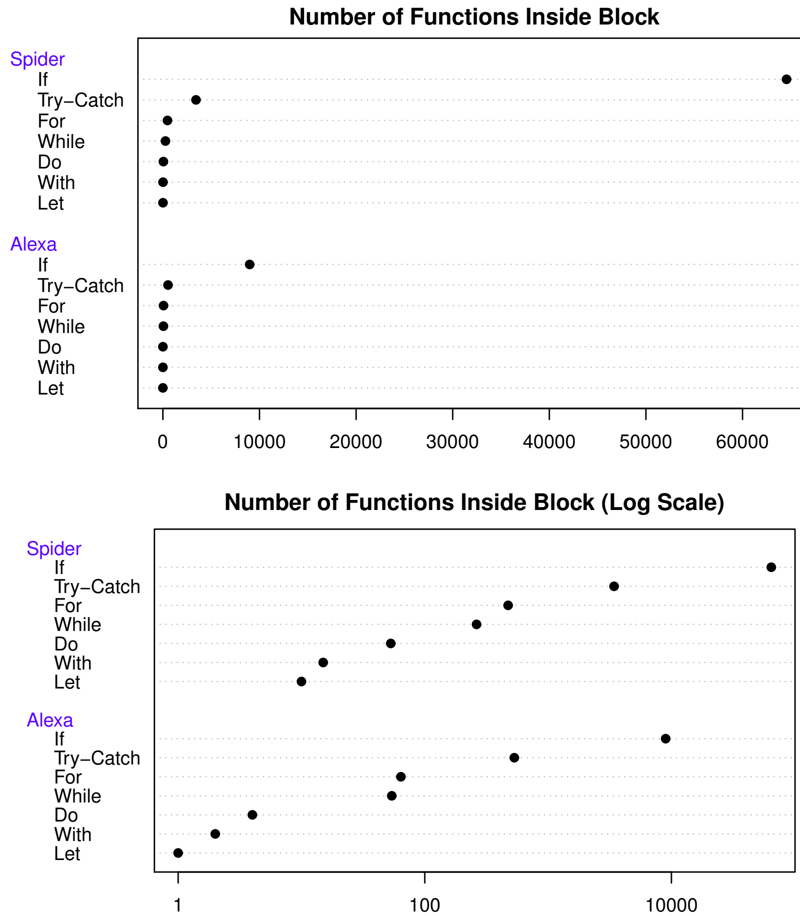

Figure 6.2: Number of functions inside block

6.4 Impact

Inner functions are used widely and performance of script suffers with incorrect usage of it. It would be worthwhile focus for optimization and should be covered more in instructional materials. It also suggests that there may be opportunities for developing tools to provide refactorings that are specific to defining and using inner functions.

The most important issue is the frequency of occurrence of inner function declarations that escape block scope. If such occurrences are rare then interpreting existing block-level function declarations as block-scoped should not have a significant impact.

The type of control structure that contains an inner function declarations also have an impact. We identified instances of functions declared in **let**. The ECMAScript is planning

to disallow the function declaration within **let**. Because, functions are not currently block scoped (function bindings are hoisted to the function level) the same function object (closure)is used no matter how many times a block is entered (during a particular out function invocation). However, block scoped declaration would create a new closure each time the block is entered. This is not a significant difference for blocks that are only entered once, such as clauses of an if statement. However, if the block is part of a looping construct block scoping will mean that each iteration of the look will use a different closure that captures different outer bindings.

Chapter 7

Objects in JavaScript

7.1 Motivation

JavaScript objects do not have the concept of private properties, although encapsulation can be indirectly achieved by declaring a variable in the function scope of a constructor and then accessing it by another function. Recent work[24] suggests adding an Encapsulate Property refactoring to JavaScript to introduce this sort of encapsulation. But the usefulness of the refactoring should be determined by whether developers need a refactoring to create private properties or whether developers use private properties at all in their code. Our work could also answer the general usage of OO concepts in JavaScript.

7.1.1 Approach

We've instrumented the interpreter part of the Firefox to check for the constructor functions. Whenever a constructor function is identified, it is flagged to verify the existence of variable declaration either by using 'var' or 'this'. These keywords represent the private and public properties respectively. The instrumented browser is used to run the JSLibraries, Alexa and Spidered pages. Mozilla Add-ons are manually checked for existence of constructor functions. We've also collected a report on usage of predefined objects by using regular expressions. We also surveyed the Node.js developers whether they used new object abstractions.

7.1.2 Results

The semantics of native objects are fully implemented in the ECMAScript standard and few examples include Date, Math, Array, Object etc. The semantics of host objects are not

Native Object	Spidered Pages	Alexa Pages	Node.js Applications
Date	145,320	18,141	1,256
Array	91,953	9,024	323
RegExp	83,249	7,896	396
Function	84,099	5,671	47
Error	62,282	6,338	1198
ActiveXObject	42,586	5,184	36
Object	21,774	2,916	34
Image	10,619	1,617	26
String	7,426	1,038	99
Map	2,095	182	13
Syntax Error	3,655	451	13

Figure 7.1: Usage of Native Objects

defined in the specification but are supplied by host to support the execution environment of ECMAScript and few of them include window, document, history, setTimeOut etc We've documented the usage of native objects and tabulated in the figure 7.1. The usage patterns are identical in both Alexa and Spidered pages. Date, Array, RegExp and Function are the widely used native objects in all the components of corpus.

7.2 Discussion

The results show that constructor functions are rarely used in JavaScript which can be used to mimic object oriented programming similar to other main stream programming language. The trend shows programmers like functional oriented style or prototypical inheritance in JavaScript. We tried to look into the way objects are created in the JavaScript by

using regular expressions by using 'new' keyword. It shows that programmers do use a lot of predefined objects like Date, RegExp etc. The usage of these objects follow a similar trend in JSLibraries, Alexa and Spidered pages.

We searched for instances of **Object.create** and found only 384 occurrences in spidered pages, and 52 occurrences in Alexa pages. None of the Firefox Add-ons pages and JavaScript libraries use **Object.create**. Private properties are rare. Private properties are rarely used not only by web developers, but also by more experienced developers of Firefox Add-ons pages and JavaScript libraries. This suggests that a refactoring to introduce encapsulated properties may not be widely adopted.

Objects can also be created by using **Object.create** which was introduced in ECMAScript 5. We searched for instances of **Object.create** and found only 564 occurrences in spidered pages, and 68 occurrences in Alexa pages. None of the Firefox Add-ons pages and JavaScript libraries use **Object.create**.

Private properties are rare. Private properties are rarely used not only by web developers, but also by more experienced developers of Firefox Add-ons pages and JavaScript libraries. This suggests that a refactoring to introduce encapsulated properties may not be widely adopted.

7.3 Impact

The results in this section along with the for...in results in Chapter 5 suggest that the actual definition of new object abstraction is rare in scripts. This coupled with the fact that Node.js developers(14 out of 34 44%) have often used new object abstractions mirrors the diversifying paradigms in which JavaScript is used. The heavy usage of new object abstractions in server-side programming justifies the effort being made to incorporate class definition syntax into in ECMAScript 6[25].

There are several possible reasons for these results. One is that programs that define object abstractions are simply not represented in the corpus. The traditional use of JavaScript

on the web was to do simple manipulations of the browser-provided DOM. Most such programs have little need to define their own object abstractions. The web-based JavaScript programs that are most likely to need to define new object abstractions are AJAX style applications that use JavaScript to manipulate complex view models or client-side Web applications that need an internal domain object model. The emergence of such programs are a relatively new phenomena and are still very rare. Given the age and size of the web, a 70,000 site sample is very small and it may be too small to include a significant number of complex applications of the type that need to define object abstractions.

Another possibility is that the approach taken in this study is not identifying the most common ways that new object abstractions are commonly defined in JavaScript. Further analysis of the corpus needs to be performed that looks for other likely object abstraction patterns.

Chapter 8

Related Work

8.1 Introduction

There have been several recent empirical studies on the dynamic properties of JavaScript. Ratanaworabhan and colleagues[11] first explored the dynamic properties of JavaScript, but their focus was to establish that the existing JavaScript benchmarks are unable to represent JavaScript code that is written. Richards and colleagues[12] had similar focus. They studied execution traces collected from 100 top Alexa pages and three industry benchmark suites to characterize the dynamic behavior of scripts and compare the dynamism with assumptions made by the benchmarks. Martinsen and colleagues[26] also found that interactive web applications (Facebook, Twitter, and MySpace) host scripts that have different execution behavior than those in benchmarks.

Our study focuses on how the language features are used and abused by programmers. Recently, Microsofts Brian Terlson studied top 10,000 Alexa pages to understand how people use strict mode, const, etc. The scope of our study and the corresponding research questions are different from his study. Even though our corpus is larger and more diversified, we chose not to repeat research questions already asked, e.g., how people use eval[13].

Researchers have been working on improving the way people write JavaScript code by exploring how to analyze JavaScript, and how to transform scripts to make them se- cure as well as elegant; many of these efforts are backed up by empirical studies to understand the JavaScript fea- tures in question. Richards and colleagues extended their previous work on understanding dynamism in JavaScript by studying the use of eval[13] and its security implications on a larger corpus (loading random pages from top 10,000 Alexa pages). This supported Meawad and colleagues[19] work on semi-automatically removing eval from

scripts. Park and colleagues[14] studied top 98 Alexa pages and 9 JavaScript libraries to find whether with statements in the scripts can be replaced with JavaScript idioms. Mirghashemi et al.[27] explored a program transformation approach to rename anonymous function studying 10 JavaScript libraries. Yue and colleagues[15] studied scripts from the top 500 Alexa pages in 15 categories (removing overlaps, 6,805 pages in total) to understand how insecure practices such as eval and innerHTML are used. They suggest that safe alternatives exist for both insecure JavaScript generation and insecure JavaScript dynamic inclusion. Nikiforakis et al.[28] studied top 10,000 Alexa pages to understand and identify the best practices of including JavaScript.

Chapter 9

Conclusion and Future Work

John Resig[29], author of jQuery has suggested that JavaScript will be treated as asignificant programming language- divorced from the concept of web development, which is the goal of the ECMAScript standard commitee.

In this thesis, we addressed the need to revamp the language and justified the necessity for the proposed 6 new features in the language. we evaluated the usage of each feature, analyzed the reasons for misconceptions in the usage of language also with the survey weve done. We believe that given the new direction the language is heading, it does need to modify its image from a just a scripting language to a mainstream language.

The next iterations of the languages are being shaped to address solutions for all the potential problems at the moment. It is called the Harmony project and while ECMAScript 6 is grearing for a spec release in early 2014 and consensus has been achieved on the implementation of few features and ther is a great deal of discussion on the implementation of certain features. Our work could be extended to throw light on other such features and genrate statistics on way the features are used in general programming community.

The other concern most of the programmers expressed is the lack of adoption of various newly introduced features across all the browsers. Mozilla firefox has very higher rate of implementation of these features in their browser. Internet Explorer showed little to no support for many of the recent features. Only few features like const, map, set have been widely implemented in all of the browsers. It would be just a matter of time while the rest of them are adopted.

Our research could be extended to some of the controversial aspects which are still pending approval for the further revisions of the language. The work could highlight the

discrepancies in the usage of the features and could be useful in arguing the need for the introduction of those features.

Bibliography

[1] Douglas Crockford. Javascript: The good parts. 2008.

[2] CompanyPR. Netscape and sun announce javascript, December 1995. URL `https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html`.

[3] Nicholas C. Zakas. *Professional JavaScript for Web Developers*. John Wiley and Sons, 2011. ISBN 9781118233092.

[4] Robert Husted and J. J. Kushlich. Server-side javascript: Developing integrated web applications, 1999.

[5] CommonJS. Javascipt:not just for browsers anymore!, November 2012. URL `http://www.commonjs.org/specs/`.

[6] ECMAScript Standards Commitee. harmony:modules, December 2013. URL `http://wiki.ecmascript.org/doku.php?id=harmony:modules`.

[7] M. de Kunder. The size of the world wide web (the internet), June 2013. URL `http://worldwidewebsize.com`.

[8] Donald. Knuth. An empirical study of fortran programs. *Software—Practice and Experience*, pages 105–133, 1971.

[9] D. Crockford. Javascript: The world's most misunderstood programming language, June 2001. URL `http://javascript.crockford.com/javascript.html`.

[10] C. Severance. Javascript: Designing a language in 10 days. *Computer*, 45(2):7–8, 2012. ISSN 0018-9162. doi: 10.1109/MC.2012.57.

[11] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. Jsmeter: Comparing the behavior of javascript benchmarks with real web applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development*, WebApps'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1863166.1863169`.

[12] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. *SIGPLAN Not.*, 45(6):1–12, June 2010. ISSN 0362-1340. doi: 10.1145/1809028.1806598. URL `http://doi.acm.org/10.1145/1809028.1806598`.

[13] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in javascript applications. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 52–78, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22654-0. URL `http://dl.acm.org/citation.cfm?id=2032497.2032503`.

[14] Changhee Park, Hongki Lee, and Sukyoung Ryu. All about the with statement in javascript: Removing with statements in javascript applications. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 73–84, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2433-5. doi: 10.1145/2508168.2508173. URL `http://doi.acm.org/10.1145/2508168.2508173`.

[15] Chuan Yue and Haining Wang. Characterizing insecure javascript practices on the web. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 961–970, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-487-4. doi: 10.1145/1526709.1526838. URL `http://doi.acm.org/10.1145/1526709.1526838`.

[16] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated construction of javascript benchmarks. In *Proceedings of the 2011 ACM International Conference*

on Object Oriented Programming Systems Languages and Applications, OOPSLA '11, pages 677–694, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048119. URL http://doi.acm.org/10.1145/2048066.2048119.

[17] Winweb Crawlerv2.0, December 2008. URL http://www.winwebcrawler.com/index.htm.

[18] Standard ECMA-262. The strict mode of ecmascript, November 2012. URL http://www.wirfs-brock.com/allen/draft-ES5.1/#sec-C.

[19] Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. Eval begone!: Semi-automated removal of eval from javascript programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 607–620, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384660. URL http://doi.acm.org/10.1145/2384616.2384660.

[20] Douglas Crockford. With statement considered harmful, April 2006. URL http://www.yuiblog.com/blog/2006/04/11/with-statement-considered-harmful.

[21] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept 1975. ISSN 0018-9219. doi: 10.1109/PROC.1975.9939.

[22] Standard ECMA-262. Section 12.1, November 2012. URL http://www.ecma262-5.com/ELS5_HTML.htm#Section_12.1.

[23] ECMAScript Standards committee. Es6 specification draft, November 2012. URL https://people.mozilla.org/~jorendorff/es6-draft.html.

[24] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for javascript. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 119–138, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048078. URL `http://doi.acm.org/10.1145/2048066.2048078`.

[25] N. Zakas. Does javascript need classes?, November 2012. URL `http://www.nczonline.net/blog/2012/10/16/does-javascript-need-classes/`.

[26] Jan Kasper Martinsen and Hakan Grahn. A methodology for evaluating javascript execution behavior in interactive web applications. In *Proceedings of the 2011 9th IEEE/ACS International Conference on Computer Systems and Applications*, AICCSA '11, pages 241–248, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4577-0475-8. doi: 10.1109/AICCSA.2011.6126611. URL `http://dx.doi.org/10.1109/AICCSA.2011.6126611`.

[27] Salman Mirghasemi, John J. Barton, and Claude Petitpierre. Naming anonymous javascript functions. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, SPLASH '11, pages 277–288, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0942-4. doi: 10.1145/2048147.2048222. URL `http://doi.acm.org/10.1145/2048147.2048222`.

[28] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 736–747, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4. doi: 10.1145/2382196.2382274. URL `http://doi.acm.org/10.1145/2382196.2382274`.

[29] John Resig. Javascript as a language, December 2013. URL http://ejohn.org/blog/javascript-as-a-language/.