A study of object creators in JavaScript

by

Yuning Yu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2017

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Traditional object-oriented languages use typed classes to specify the shape and behaviour of objects. In JavaScript, object behaviour is less constrained. Learning how JavaScript programmers create objects can help us understand whether the programmers employ the notion of “classes”, like those in Java, in JavaScript. In other words, we are trying to compare the creation of objects in JavaScript with Java classes, which create objects and describe the state and behaviour that the type of the object support. Our results have the potential to improve the performance of JavaScript and generality of typed extensions to JavaScript.

An access site is the JavaScript code that accesses the properties of objects. We studied the access site to determine whether objects at the access site arise from different “creators”, which are functions creating the objects, and to discover how prevalent multiple-creator access sites are. Moreover, whether the types of properties ever change is noteworthy: in classes, types of properties may not change. To answer the questions, we executed, recorded, and analyzed many JavaScript applications to obtain real-world data.

We collected and analyzed logs from 7,753 web sites. The logs consist of nearly 390 GB of JSON strings. We collected these data through dynamically running modern JavaScript applications, or web sites, in our tool with the browser Firefox.

Our results show that for half of selected web sites, there exists only one creator at every access site. And in fewer than 50% of web sites, there exists more than one creator, though the total number of access sites with multiple creators is extremely small. Although creators with distinct names are considered as different by our definition, different creators may create same objects. Therefore, most object accesses in JavaScript do not come from different creators. As a result, we could employ type annotations for objects.

However, our results show that the types of the properties in objects do sometimes change, especially in objects generated from user-defined functions. Thus, even if objects are mostly from one creator, that creator may not correspond neatly to a type. This may limit the advantages of using types for better performance.

Combining these results, we conclude that while we can apply Java-like types to JavaScript, these types are far more fluid in JavaScript applications, and so there may be limited benefit.

Acknowledgements

First, I would like to thank my supervisor Prof. Gregor Richards for the support of my master study and research. Whenever I ran into troubles, he would always offer helps. He always allowed the thesis to be my own work, but directed me on the right road.

Besides my supervisor, I would like to express my gratitude to the rest of the thesis committee: Prof. Peter Allan Buhr and Prof. Patrick Lam, for the valuable comments, as well as many discussions, to revise the thesis.

Last but not the least, I would like to thank my family, who have supported me through the entire process.

Table of Contents

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Background and Motivation 1

1.2 Introduction of Methodology 4

2 Related Works 8

3 Methodology 12

3.1 Infrastructure 14

3.2 DotInstr 14

3.3 Analysis Tool 20

3.4 Recorded Data 25

3.5 Limitations 28

4 Observations 30

4.1 Number of Creators 30

4.2 Category of Creators in Single and Multiple Access Sites 33

4.3 Types of Properties in Objects 37

4.4 A Discussion of Creators on “Partial Read Only” Web Sites 40

4.5 Statistics of Unstable Writing Operations 45
4.6 Case Analysis 46
4.6.1 Analysis of Unusual Web Sites 46
4.6.2 Analysis of Source Code from Three Examples 48
4.7 Web Sites that Break DotInstr 54
4.8 Global Fields 54

5 Conclusion **55**

References **57**

APPENDICES **59**

A JSON Data Extracted for Case Analysis **60**
A.1 A Detailed JSON Data 60

List of Tables

1.1 Classifications of possible patterns. 7

3.1 Classifications of web sites. 23
3.2 Recorded data about access site. 25
3.3 Recorded data about creators. Figure 3.6 generates the data. 26

4.1 Almost all property access sites have a single creator. 31
4.2 Numbers of single and multiple creators web sites are both substantial. 31
4.3 Most web sites contain single and two creators access sites. 33
4.4 Access site with single-creator is the majority. 33
4.5 “read only” creators are much more than others. 38
4.6 Partial read-only web sites are the majority. 39
4.7 Definitions of operations. 40
4.8 Three stable built-in creators most commonly used. 44
4.9 The creator `Object` is unstable. 45

List of Figures

3.1 The recording and analysis processes at the high-level. 13
3.2 Comparison of opening web sites with and without DotInstr. 15
3.3 Instrumented elements of web sites through DotInstr. 16
3.4 The process about the modification of JavaScript code. 21
3.5 Detailed analysis process. 24
3.6 Sample code that generates the data in Table 3.3. 26

4.1 The rate of built-in creators to multiple-creator is low in most web sites. 36
4.2 There exist substantial dynamic creators. 41
4.3 Built-in creators account for a small portion in dynamic ones. 43
4.4 Rate of unstable writing are much more than stable ones. 47

Chapter 1

Introduction

1.1 Background and Motivation

JavaScript has become a widely used language, particularly in web programming, due to the support by web browsers. It is a highly-dynamic object-oriented programming-language; everything, including object properties and even an object's prototype—equivalent to its class in other object-oriented languages—can change at run-time. These dynamic features bring challenges to static analysis. Research on JavaScript concentrated on three fields: performance, security, and correctness. This thesis focuses on the type of objects, which has influence on performance and correctness.

We take the creator of objects as the type. In this thesis, the creator of the object is defined as the function in which the object is created. There are three general ways to identify the creator, which could be demonstrated by the following code.

(i)
```
function foo(){
    var obj = {k1:1};
}
```

The creator of `obj` is `foo`.

(ii)
```
function foo(){
    var obj = new Object();
}
```

The creator of `obj` is `Object` because `obj` is generated by the constructor function `Object`. `Object` is a built-in function that cannot be modified. However, like any JavaScript object, `obj` may have new fields added later.

(iii)
```
function Person(firstName, lastName) {
    this.firstName = first;
    this.lastName = last;
}

function foo(){
    var obj = new Person("John","Smith");
}
```

The creator of obj is the function `Person`.

We associate each object at run-time with its creator. It is then possible to group objects created by the same creator and distinguish among objects created by different creators, and thus to count them for analysis.

We should mention that in results based on the creator, there is a special creator named the "global field", which is the root field in source code. The creator of objects created globally is recorded as `global field`. For example, in a source-code file with the following code:

```
// root field
var obj1 = {k1:1};

function foo(){
    // inside a function
    var obj2 = {k2:2};
}
```

the creator of `obj1` is `global field`, while the creator of `obj2` is a normal creator, `foo`.

When counting different types of creators, the results exclude this special creator. The creator `global field` may affect the accuracy of final results, since the size of `global field` is much greater than other creators, but it is clearly unreasonable to treat global objects identically to non-global objects.

The core question of this work is whether types akin to Java-like "classes" are applicable to JavaScript. We approach this problem through object creators: Java classes are nominal, and thus any object of a type was, by definition, created by that class or a descendant. We therefore expect that, if real JavaScript code behaves similarly, the set of objects created by any function, herein the creator, should behave as a class-like type.

We accomplished our works through the analysis of logs recording the execution of JavaScript. In principle, if objects in JavaScript behave like classes, we can take advantage of the analysis result and import types to perform better on looking up properties. Inline caching [1] is a well-known example among existing techniques in improving performance of JavaScript. Our result aims to establish an empirical basis for its effectiveness.

The type of objects is regarded as stable if the objects exhibit class-like behavior. The type contains information about the address of the property values. Inline caching stores the type of objects and looks up properties quickly through the saved type. Looking up properties would save considerable time if most objects are of stable type.

For each property access site, i.e., each location in JavaScript code that accesses a property of an object, such as `Object.property`, if there is only one creator of the object, and the types of properties do not change, a JavaScript engine could assume that the type of the object is stable. If this were known soundly, JavaScript could speed up accessing the properties. For example in the following code:

```
function foo(obj){
    obj.method();
}
```

when `foo` is called, the runtime needs to know both the type of `obj` and how the function `obj.method` is implemented.

Importing types to JavaScript improves the effectiveness for searching a property value. Modern JavaScript engines save object properties in a `shape` and an `array` or `store`. The `shape` maps properties to their corresponding values in the `array`. As a result, the runtime needs to look up the `shape` first, then the value in an `array`. However, if objects occur at a particular site always, or at least often, belongs to the same `shape`, it is possible to improve the performance by caching and using the `shape` directly instead of searching for the `shape` first. Objects with the same type refer to a same `shape`. Provided there is a type system in JavaScript and we observed that `obj` follows the mentioned pattern, i.e., always the same type. Assuming the type is `String`, we could revise the previous example as the following one.

```
function foo(String obj){
    ((String)obj).method();
}
```

Instead of searching for the type of `obj`, we use the cached type `String` as a `shape` to get the implementation of `method` in a known `array`. Since the runtime knows

the `method` and does not need to look up it dynamically, the performance is greatly improved due to types.

1.2 Introduction of Methodology

To measure object behaviour in JavaScript, the first approach was instrumentation of JavaScript engines such as Mozilla's SpiderMonkey, used in Firefox, and Google's V8 engine, used in Chrome. However, functions that we want to instrument call deeply into other layers outside of C++ (the programming language used to implement the two engines). Both engines use native property access in the just-in-time (JIT) compilation, and neither of them implements an interpreter. JIT indicates that code including property access is compiled during run time instead of prior to execution. This makes instrumentations of both engines extremely difficult.

The second approach, what we used in our work, is to instrument through a proxy. A proxy is a service that accepts transactions from clients, processes transacted data, and sends the data to a server or browser. Hence, a proxy between web sites and the browser could also perform measurements of the object since it is accessible to all JavaScript code. And as mentioned in the previous paragraph, we cannot easily instrument both of the two JavaScript engines in their latest versions. So we chose the proxy approach.

In our work, we employed the proxy underlying JSBench [10]. JSBench offers a solution to create JavaScript benchmarks using JavaScript itself, through recording and replaying real JavaScript web sites. JSBench acts as a proxy between the web site and the browser. It modifies JavaScript code such that when events of user-interactions occur in a standard web-browser, JSBench records the events. After retrieving the related code, JSBench parses code into an abstract syntax tree (AST), which is then modified to add customized instrumentation. We can describe the recording process of JSBench as follows.

(i) Retrieving and parsing JavaScript code into an AST.

(ii) Modifying the AST nodes.

(iii) Making the browser send required data back to the proxy for recording.

We modified the AST nodes through a new tool named DotInstr, which records accesses to object properties. We built it on the JSBench, but added new instrumentations and data recording. In spite of some slowdown in the recording process, DotInstr contributed to reduce time consumption in total, since it recorded all necessary data once instead of multiple times. Through DotInstr, we only need to perform the recording process once. Besides, we could run as many rounds as necessary of the analysis process. The analysis process only costs less than 5% of the time needed for recording. DotInstr worked similar to a record-and-replay paradigm, which is a kind of dynamic analysis.

The reason for adopting dynamic analysis is that static analysis tools fail to represent the dynamic side of modern JavaScript applications. Overly conservative results from these static analyses, like that from Silva *et al.* [12], may not be useful for representing the dynamism of JavaScript.

By design, DotInstr focuses on two cases of JavaScript code. For one case, after the creation of the object, DotInstr saves the creator into the object as a new property. For the other case, when accessing the properties of the object, DotInstr records the creator saved in this object and where the accessing occurs. Through the recording, DotInstr generates log files for analysis. The analysis process works after the recording process by another tool to produce results.

We can perform different analyses with a single captured log, instead of recording multiple times for each behaviour that needs to be measured. In our work, the recording process took more than one week. But the analysis tool only used hours to finish working on the recorded files.

DotInstr can find several possible patterns in modern JavaScript applications, shown in Table 1.1. There are examples to explain these patterns.

(i) Object creation is Java-like:

```
function bar(name){
    this.name = name;
}

function foo(obj){
    // 'obj' is from the same creator: 'bar'
    console.log(obj.name);
}

var obj1 = new bar("A");
foo(obj1);
```

```
var obj2 = new bar("B");
foo(obj2);
```

Object creation is JavaScript-like:

```
function bar(name){
    this.name = name;
}

function model(name){
    this.name = name;
    this.size = 10;
}

function foo(obj){
    // 'obj' is from two creators: 'bar' and 'model'
    console.log(obj.name);
}

var obj1 = new bar("A");
foo(obj1);
var obj2 = new model("B");
foo(obj2);
```

(ii) Properties of object are Java-like:

```
function Person(firstName, lastName) {
    this.firstName = first;
    this.lastName = last;
}
var obj = new Person("John","Smith");
// object behaviours do not change types of properties
obj.firstName = "Aaron";
```

Properties of object are dynamic:

```
function Person(firstName, lastName) {
    this.firstName = first;
    this.lastName = last;
}
var obj = new Person("John","Smith");
var tmp = obj.firstName;
obj.firstName = "Aaron";
// properties of the object 'obj' change during execution
obj.usedFirstName = tmp;
```

The outcome of this classification should help to make decisions on whether types in JavaScript can be treated like those in Java. There is a possibility that some applications contain class-like objects, while others do not. The analysis of JavaScript applications that do not use class-like objects is also valuable. For example, if we found that these applications use a similar third-party library, then the analysis of the library is of value.

Table 1.1: Classifications of possible patterns.

Java-like Object Creation	When accessing an object, there exists only one creator of the object.
JavaScript-like Object Creation	There exists more than one creator when accessing an object.
Java-like Object Properties	After object creation, the types of properties do not change.
Partial Java-like Object Properties	After object creation, the types of properties do not change. There are also logs showing that the types and existence of properties changed after creation.
Dynamic Object Properties	The properties of object could change after creation.

Chapter 2

Related Works

Numerous researches related to our work cover two directions generally: static analysis and dynamic analysis. These techniques inspired the methods that we used in this thesis.

(i) Static analysis of JavaScript focuses on looking for patterns in JavaScript code. Silva *et al.* [12] proposed a strategy to find classes in JavaScript code. They focused on prototypes that are used as a template for the creation of objects. The paper analyzed 50 JavaScript applications on GitHub and classified them into 4 different groups based on their usage of classes. However, their work relied on the keyword `new` to detect object creation. This strategy did not cover frameworks that generate objects in different ways, like Angular JS. Additionally, this paper did not deal with dynamic changes, like object may change during execution, except giving a warning. Ocariza *et al.* [7] analyzed more than 300 bug reports, from eight web applications and four JavaScript libraries, to figure out the cause of JavaScript faults. Their results indicated that most bugs are caused by errors in interactions between JavaScript with DOM (Document Object Model). However, these bug reports were generated from limited number of experimental objects, which may lead to questions on the representativeness. To avoid these mentioned issues, our subject is not on whether class-like types can be discovered statically, but whether real code exhibits class-like behaviour.

(ii) A dynamic analysis framework, on the other direction, inspired us to employ record and replay paradigm. Sen *et al.* [11] built a framework, Jalangi, for

JavaScript dynamic analysis. Jalangi utilized a record-replay mechanism to allow heavy-weight analysis. Since Jalangi employed a method to copy memory during the execution of JavaScript, it could record all memory loads during the interaction between the browser and web sites. As a result, Sen *et al.* analyzed plenty of data.

But the implementation of Jalangi did not work on some special cases, such as strict mode and updating properties by setter methods of the object. Jalangi relies on the function `arguments.callee` to get the necessary information, but some JavaScript engines forbid the use of the function in strict mode. It is likely that our work could have been implemented based on Jalangi and would have had the same results. Unlike Jalangi, DotInstr works well in strict mode. DotInstr checks the key word "use strict" and removes the key word in the source code to disable the strict mode, thus allows `arguments.callee`. However, our work of DotInstr also has its own limitations shown in Section 3.5. And DotInstr does not work on setter functions either, since our focus is the properties accessed by dot notation.

(iii) We also learned from other dynamic analyses employing behaviour recording methods. The following papers are all based on behaviours of real-world JavaScript applications.

(a) Ratanaworabhan *et al.* [9] evaluated and compared behaviours between JavaScript web applications and JavaScript benchmarks, by instrumenting the IE8 runtime. Their paper employed V8 benchmarks from Google and part of SunSpider from WebKit.org. Ratanaworabhan *et al.* concluded that these benchmarks were not representative of popular web applications and might be misleading. According to the paper, the instrumentation of the JavaScript engine is one effective approach. Thus we also tried to instrument the V8 and SpiderMonkey JavaScript engines.

(b) Nikiforakis *et al.* [6] evaluated the ability of JavaScript to include libraries from remote resources to a local page. Through a web crawl, Nikiforakis *et al.* recorded and analyzed the relationship between top web sites and their service providers. They reported that the service providers could be compromised by attackers in some cases, thus provide malicious JavaScript. Nikiforakis *et al.* worked on more than three million pages of the top 10,000 Alexa web sites, which was more accurate than our data based on the number of pages that they crawled.

(c) Gude *et al.* [4] studied how JavaScript language features were used by programmers in practice. This paper employed an instrumented SpiderMonkey JavaScript engine in Firefox to record data. Through recorded data, the paper focused on the use of strict mode, the "with" keyword, scopes of variables, functions inside blocks, enumeration with for...in, and objects in JavaScript. Gude *et al.* found that there are confusions and misuses about new features, and the lack of object-oriented programming style in JavaScript. Their work may indicate that the application of types to JavaScript is limited.

(d) Pradel *et al.* [8] dynamically analyzed JavaScript execution by a tool built based on Jalangi [11], to study the type coercions. The tool instrumented JavaScript similarly to our work. It created a file on local disk for each instrumented JavaScript program. Their result showed that type coercions are widely used and harmless. The result indicates that type coercions may also limit the application of types to JavaScript.

Comparing these related papers, we find that static analysis tools do not work as well for JavaScript as for other languages. As Livshits *et al.* [5] suggested, static analysis tools choose not to handle complex language features that would make the tools imprecise.

Unlike static analysis, behaviour recording has proved to be successful and efficient to deal with the dynamic features of JavaScript. The recording process can capture the execution of JavaScript applications, and thus is more precise than static analysis if the tool record all required behaviours. Based on the comparison, we employed behaviour recording as the approach to perform measurements.

Besides these related papers, some state-of-the-art techniques focusing on performance improvement through types are noteworthy as well.

Modern JavaScript engines like V8 utilize inline caching. As mentioned, inline caching uses the cached type of the object to find the property values. This thesis attempts to prove the intuition behind it. Inline caching is based on the prediction that while accessing a property of a given object, types of all future objects at this site are the same, to achieve better performance. If the future type does not match the cached type, the runtime will need to look up the location that stores values of properties with extra efforts. Our work provides observations to suggest that the prediction is correct that most future type is same with the cached one.

Instead of identifying the creator, V8 JavaScript engine defines the "hidden class"

on each object to record the location about values of properties in memory. In the following code, after `this.val2 = val2` is executed, the hidden class saves the offset to find the property `val1` and `val2`.

```
function Obj(val1, val2){
    this.val1 = val1;
    this.val2 = val2;
}

var foo = new Obj(1,2);
```

This approach is more strict than our definition of creator, as updating values of properties in a different order could generate different hidden classes. However, both of them are based on the assumption that objects may exhibit class-like behaviour.

Another similar work is from optionally typed languages, in particular TypeScript [3]. TypeScript provides support for types and classes in JavaScript. It is a superset of JavaScript. The static typing guarantees in principle that there is only one creator—the class—used at each property access site. However, TypeScript is an intentionally unsound type system that no mechanism satisfies the previous guarantee. In early versions, TypeScript employed nominal typing. The nominal typing system considers types with the same name as the same type. However, recent versions of TypeScript have employed a structural typing system, in which types are deemed to be the same if they have the same structure. This change was made based on the developers' intuition with no evidence from real programs. Our results predominantly contradict this decision, since most property access sites only have one creator and we distinguish the creators by their names. This observation motivated the particular experiments performed to get the results in Chapter 4.

Chapter 3

Methodology

This chapter describes the methodology that we used to collect and analyze data.

This thesis aims to gather data to determine whether JavaScript objects behave as if types exist. In particular we expect to observe whether objects are always from one creator, and whether types of fields in objects change. Dynamism of JavaScript allows programmers to modify these items during execution. The goal is therefore not to answer what *could* happen, but what *does* happen. Static analysis tools are able to predict the execution of JavaScript, but not able to reveal what really happened. To this end, we employed dynamic tools to collect data. When launching a browser and visiting a web site, transactions happen: the browser will execute code from the web site. In this process, it is possible to intervene with a standalone program, which captures information about JavaScript execution, to achieve the statistics about the creators and the type changing for each JavaScript object. To collect relevant information, we experimented on the top 10,000 web sites as designated by Alexa. To launch these web sites with modern JavaScript engines, we selected Firefox as the web browser.

Our tool DotInstr opens a web site and stays on the home page for one minute. During this period, DotInstr collects data generated from JavaScript executions and saves the data to a single log file. DotInstr repeats the same process on all web sites automatically. After recording all web sites, an analysis tool starts to analyze the log files. Figure 3.1 shows the recording and analysis processes at the high-level. The steps can be elaborated as follows.

(i) The automation script runs DotInstr and opens the browser with URL at the same time. JavaScript code from the web site is then loaded to DotInstr.

(ii) DotInstr instruments the JavaScript code.

(iii) DotInstr sends the modified code to the browser.

(iv) The browser sends back data to DotInstr for recording.

(v) DotInstr records the data to a local file.

(vi) The automation script repeats the previous steps until recording all web sites.

(vii) The automation script stops.

(viii) The analysis tool starts to examine the recorded data and generates result.

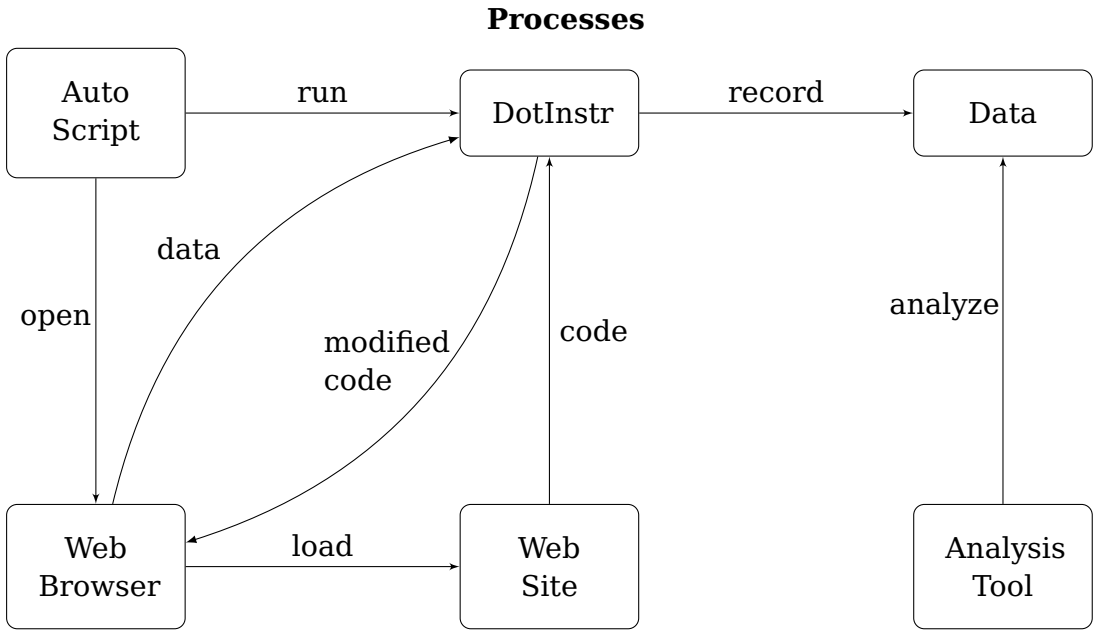

Figure 3.1: The recording and analysis processes at the high-level.

The size of the data we recorded is about 390GB. These data are generated by automatically opening each web site, waiting for the home page to finish loading and showing the contents, then closing the browser. For a better accuracy, the recording process is similar to the routine way people open web sites. We tested on 100 web sites to make sure functionalities of the web sites work well through DotInstr. We did not check whether all web sites work well in the recording process.

Finally, an analysis tool gets the results from these logs, by rebuilding how relevant events happened during execution and counting creators and type changes. The analysis process is similar to recording but much simpler: through the automation script, the analysis tool works on all recorded files one by one and then generates the result.

3.1 Infrastructure

As mentioned, DotInstr aims to capture JavaScript executions including object creating and property accessing when the browser and web sites communicates.

Figure 3.2 shows how web access works with and without DotInstr. “Web site” is a set of web pages. Normally, the bowser loads and displays web pages directly. Viewing web pages through DotInstr is a different case, as DotInstr interjects between the browser and the web site. As we know, a web page consists of four types of elements: HTML, scripts, styles, and multimedia. DotInstr, which focuses on JavaScript executions, involves code in script tag in HTML and all script elements. Figure 3.3 illustrates a standard instrumentation approach. In Figure 3.3, the four nodes on the left side represent the four kinds of elements. Then DotInstr instruments JavaScript code in HTML and scripts files. So DotInstr modify the two nodes representing HTML and scripts files, and change them to two new nodes with a different color. At last, all elements reach the web-browser.

When opening the browser with URLs, DotInstr acts as a proxy between the browser and web sites. It parses JavaScript code to abstract syntax trees (ASTs), and modifies them at that level. The proxy modifies nodes representing property access site. The revised property access performs the relevant access, but also record information to the local disk. DotInstr records data in JSON. The analysis tool examines the JSON data off-line from the local disk, and reports the analysis result. The analysis tool calculates all information discussed in Chapter 4.

3.2 DotInstr

As described above, DotInstr acts as a proxy between the browser and web sites. Recording consists of a server part, which runs in the proxy itself, and a client part,

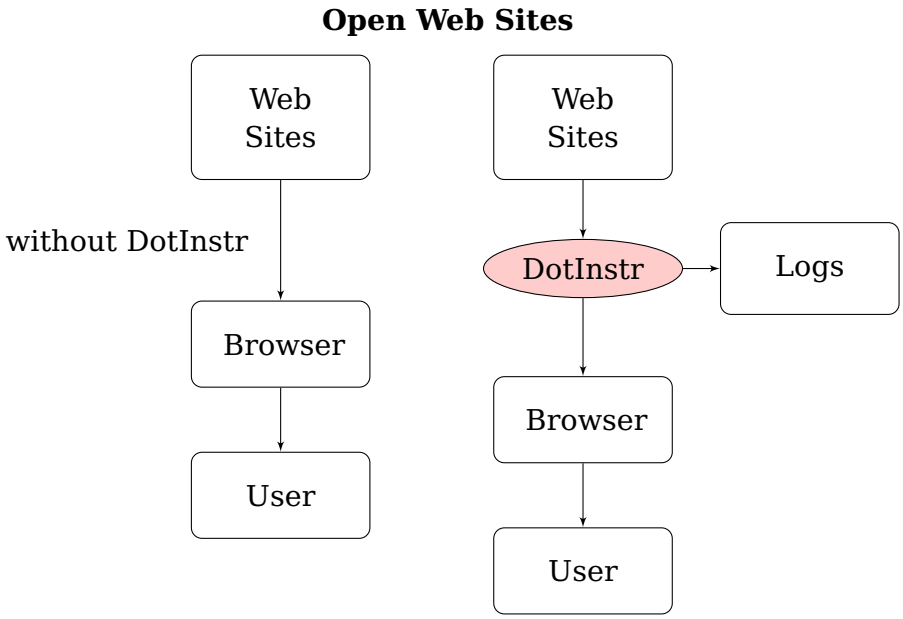

Figure 3.2: Comparison of opening web sites with and without DotInstr.

which runs in the browser. The bowser treats the modified code identically to that from a normal web site.

When object is created in JavaScript code, DotInstr adds a new property, `_jsb_creator`, to the object to save the name of creator.The reason why we need to save the name of creator during object creation instead of property accessing is that DotInstr cannot locate the creator of object at the property access site. When accessing the property, DotInstr stores the aforementioned creator-name and the type of property in a JSON item. It also records the URI of the source-code file and position of the access site in the same JSON item.

The pseudocode for adding creators and recording logs follows.

(i) Original code is:

```
x={};
```

After modification, DotInstr adds an un-enumerable property in the object `x` to save its creator:

```
x={};
x._jsb_creator = creator;
x._jsb_creator.enumerable = false;
```

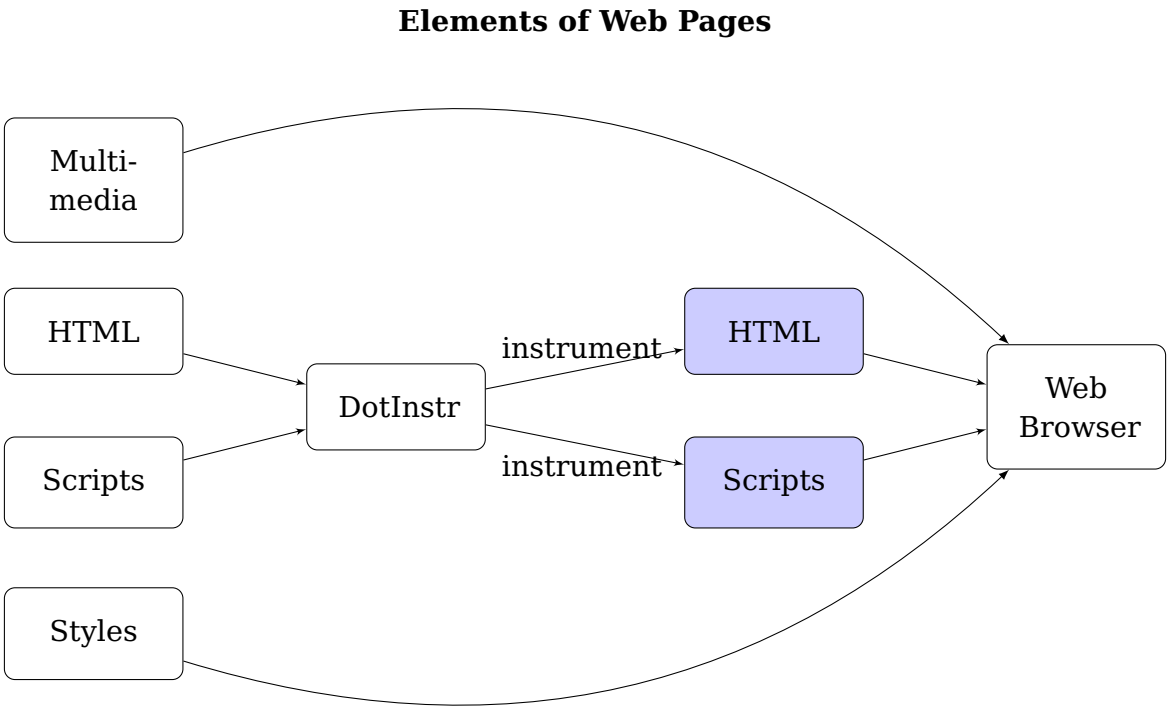

Figure 3.3: Instrumented elements of web sites through DotInstr.

Setting `enumerable` to `false` is to avoid `_jsb_creator` to affect normal code. `for-in` loops and other reflective features in JavaScript discover only enumerable fields. For example, if the programmer creates a new object based on all the data fields in the object, it may cause problems if `_jsb_creator` is set to enumerable:

```
var x={baseUrl:"http://www.serveraddress.com", api:"/userInfo", id:"/uid001"
    };
// Through DotInstr, it adds the following code to original code:
// x._jsb_creator = "global field";
var url;
for(var property in x){
    // concatenation
    url+=x[property];
}

var xhr = new XMLHttpRequest();
xhr.open("GET", url, false);
xhr.send(null);
```

The variable `url` is `http://www.serveraddress.com/userInfo/uid001/global field` that elicits a different result from the original code.

(ii) Original code is:

```
x.y;
```

After modification, DotInstr adds a function following `x.y` to record logs to local disk:

```
x.y;
record(x,y,...);
```

The following sample explains the process of modification in detail.

During object creation, DotInstr employs three functions for adding the property which saves the creator. The functions `instrumentNode` and `addCreator` run in the proxy side of DotInstr for each object creation-site. The function `addCreator` modifies the AST. With the modification, the browser side of DotInstr calls the function `makeObjCreator` to add the property.

```
function instrumentNode(n){
    if (n.type === OBJECT_INIT){
        n = addCreator(n, TYPE_BRACKET_OBJECT);
```

```
    }

    if (n.type === NEW_WITH_ARGS){
        n = addCreator(n, TYPE_NEW_OBJECT);
    }
}

function addCreator(n, type) {
    // condition checks by 'type' is omitted for readability
    return fakeNode(n, CALL, [
        fakeNode(n, IDENTIFIER, null, {value: "JSBNG_Record.makeObjCreator"}),
        fakeNode(n, LIST,
            [n],
            {parenthesized: false})
    ]);
}

function makeObjCreator(ret) {
    Object.defineProperty(
        ret, '_jsb_creator', {
            value: (
                (makeObjCreator.caller === null ||
                (makeObjCreator.caller).toString() === "") ?
                "global field" : makeObjCreator.caller.name
            ),
            enumerable: false
        }
    );

    return ret;
};
```

To save information of the property access site, there are three functions in DotInstr to modify dot expressions (standard property access). DotInstr calls the function `dotToCreator` on each property access site to modify the corresponding AST for instrumentation. This function sends all necessary data, including the name of object, the property, the URI of source-code file, the line number of code, the type of property and new value if any to the browser side of DotInstr.

Then on the browser side, function `recordDot` is executed. After building a JSON item including both property access site and extra fields about type changes, the function sends the JSON item back to proxy side and makes the proxy save logs into local file system. Additionally, the `recordDot` function executes the original code of the property access. Parameters used in `recordCreator` are extracted by

the arguments object. We omit the process of extracting parameters in the code demonstration for readability, which is why the argument list is empty in `recordDot`. Among the arguments, `isWrite` depends on the `type` field of AST node. For instance, if `type` is ASSIGNMENT, `isWrite` is resulted to be true while `type` DOT to be false. `propertyType` is the type of the property before a writing operation. `newType` is the type of the property after the writing operation. And `isNewProperty` is true if adding a non-existent property to the object.

```
function instrumentNode(n){
    if (n.type === DOT){
        n = dotToCreator(n);
    }
}

function dotToCreator(n) {
    return fakeNode(n, CALL, [
        fakeNode(n, IDENTIFIER, null, {value: "JSBNG_Record.recordDot"}),
        fakeNode(n, LIST, [
            fakeNode(n, STRING, null, {value: variableName}),
            variable,
            fakeNode(n, STRING, null, {value: property}),
            fakeNode(n, STRING, null, {value: variableNode}),
            fakeNode(n, STRING, null, {value: fileName}),
            fakeNode(n, NUMBER, null, {value: variable.lineno}),
            fakeNode(n, NUMBER, null, {value: variable.start}),
            fakeNode(n, NUMBER, null, {value: variable.end}),
        ], {parenthesized: false})
    ]);
}

function recordDot() {
    var accessor = (recordDot.caller === null? "global field":
    recordDot.caller.name);
    recordCreator(variableName, variable, property, varNode,
        accessor, fileName, lineno, start, end, propertyType, isWrite,
        isNewProperty, newType);
    return (variable)[property];

};
```

Further, according to whether they write to the property and their own type behaviour, we distinguish the property access site to assignments, increments, decrements, method calls, etc. DotInstr replaces the access itself with a call to a function for adding creators and recording logs. For each type, DotInstr adds required argu-

ments to the function to keep the original meaning of the code.

We realize the separation of property access site through the field `type` in each AST node. Take the assignment `x.y = 1` as an example. As `x.y` is not only accessed but also changed, DotInstr replaces the entire assignment node representing `x.y = 1` with a call node. DotInstr changes the type of this AST node from `ASSIGNMENT` to `CALL`. The `CALL` indicates a call to our customized function replaces `x.y = 1`. DotInstr substitutes a unique function for different styles of property access site accordingly. In this case, DotInstr calls the method `recordDotAssignment` in the browser, with arguments: x, y, 1, ASSIGNMENT, instead of `x.y = 1`. `recordDotAssignment` is aware of the assignment operation in the AST node by the arguments. The function assigns `1` to `x.y`, as well as building a JSON object to record this action. Finally, it sends the JSON object back to the proxy side of DotInstr for saving the data to the local file system.

Figure 3.4 clarifies and explains how DotInstr instruments code. Nodes in blue color are action nodes. The process is as follows:

(i) Recall that we built DotInstr on JSBench. JSBench parses JavaScript code to an abstract syntax tree. So the parser in Figure 3.4 parses original code to AST.

(ii) DotInstr checks whether the node is a target node: object creation or dot notation.

(iii) DotInstr modifies the target AST node as described in this section.

(iv) DotInstr replaces the target AST node with the modified one.

(v) Through JSbench, it turns modified AST back to modified JavaScript code by the serializer in Figure 3.4.

3.3 Analysis Tool

After the recording process, the analysis tool starts to analyze the data off-line. We developed the analysis tool in Java. It aims to count the number of sites and creators aforementioned in each log file, and summarize the results for Chapter 4. The work flow for counting is as follows (in a simplified version without error-checking code like null pointers):

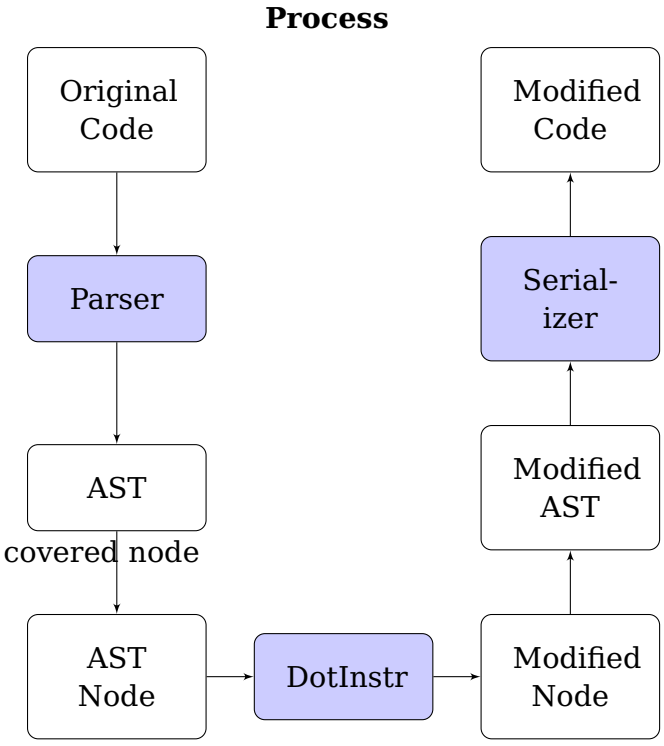

Figure 3.4: The process about the modification of JavaScript code.

```
private void handleStr(String str){
    // 'JSRecordObj' contains all recorded data at a property access site
    JSRecordObj propertyAccessSiteInfo = new Gson().
        fromJson(str), JSRecordObj.class);
    recordSiteCreators(propertyAccessSiteInfo);
    recordObjPropertyChanges(propertyAccessSiteInfo);
}

private void recordSiteCreators(JSRecordObj propertyAccessSiteInfo) {
    // 'mapSiteToCreators' is to save following information:
    // key: property access site
    // value: a collection of creators found on the access site
    HashSet setOfCreators = mapSiteToCreators.get(propertyAccessSiteInfo._site);
    setOfCreators.add(propertyAccessSiteInfo._creator);
}

private void recordObjPropertyChanges(JSRecordObj propertyAccessSiteInfo){
    if(propertyAccessSiteInfo._isWrite &&
```

```
            propertyAccessSiteInfo._isOutsideCreator){
        if(propertyAccessSiteInfo._isNewProperty){
            mapCreatorToChanges.
                get(propertyAccessSiteInfo._creator).incrementNewFieldsChange();
        }else if(propertyAccessSiteInfo._isTypeChange){
            mapCreatorToChanges.
                get(propertyAccessSiteInfo._creator).incrementWriteChange();
        }else{
            mapCreatorToChanges.
                get(propertyAccessSiteInfo._creator).incrementWriteNoChange();
        }
    }
}
```

The analysis script first uses the command `cat` to read the content of a log file. Then the analysis tool reads the content line by line into memory. As mentioned, DotInstr saved logs in JSON, the function `handleStr` converts each line of JSON string to JSON object. A line in the log file represents a property access site in JavaScript code.

The function `recordSiteCreators` saves the site and creators for each JSON object.

In the last function, `recordObjPropertyChanges`, it inspects whether there are type changes or new added properties in objects under each creator. It also checks type-stable assignments, to record whether there is a new value assigned without type modification. It should be noted that the analysis tool checks type changes outside of the creator, since inside the creator, simply adding the property invokes a type change. `JSRecordObj` contains the values `isWrite` and `isNewProperty`. The parameter `isOutsideCreator` denotes whether the creator equals to the function that accesses the property.

Additionally, since DotInstr saves information in JSON, the analysis tool relies on a library, which is Google's Gson in this work, to convert between JSON strings and JSON objects.

After traversing the entire log file, the analysis tool uses the two recorded `Maps`, `mapSiteToCreators` in `recordSiteCreators` and `mapCreatorToChanges` in `recordObjPropertyChanges`, to produce results.

In the first `Map`, `mapSiteToCreators`, key is the location of the property access site, and value is a set of creators found on this site. The result informs how many different creators are found on each property access site, as well as the occurrences

of different cases (one creator, two creators, three creators, etc). With the knowledge of whether there are multiple creators and which case is the majority in a web site, we can learn the JavaScript applications in a more dedicated way.

Second `Map`, `mapCreatorToChanges`, takes the name of creator as the key, and numbers of changes as the value. This map computes the number of writing operations for objects created by the creator. Furthermore, the map provides the amount of type changing in writing operations and thus the number of creators with type changing. We classified web sites into five types respectively as shown in Table 3.1. We used them to show the stability of creators in a web site in general. For example, the "all read-only" web site tells us that all creators, except global field, are stable with no change. After that, by counting the creators with different classifying methods, which are explained in Chapter 4, and by extending the analysis tool, we can provide more detailed analyses in various aspects.

Table 3.1: Classifications of web sites.

Type	Description
all read-only	All fields have never been written.
partial read-only	Some fields have never been written, others have.
all type-stable	Types of fields have never been changed.
partial type-stable	Types of fields have been changed, with others have not.
all dynamic	Types of fields changed without patterns.

A Python script runs the analysis tool automatically on all web-sites log-files and saves the analysis results to a local file in which each line represents the result of a web site, in the format of JSON.

After getting the file containing result, we execute a stand alone Java program to count the occurrences (access sites) in each case of different number of creators, as well as the types of creators, among all web sites.

The work flow of the analysis process is in Figure 3.5. The analysis tool extracts data from each JSON object and saves them in the three blue nodes. Combining property access site and creator, the analysis tool generates the result about the number of creators on each property access site, which in Figure 3.5 is represented by the red node "Result 1". Similarly, through creator and change of properties, the analysis tool gets the result of creators, which is represented by the red node "Result 2".

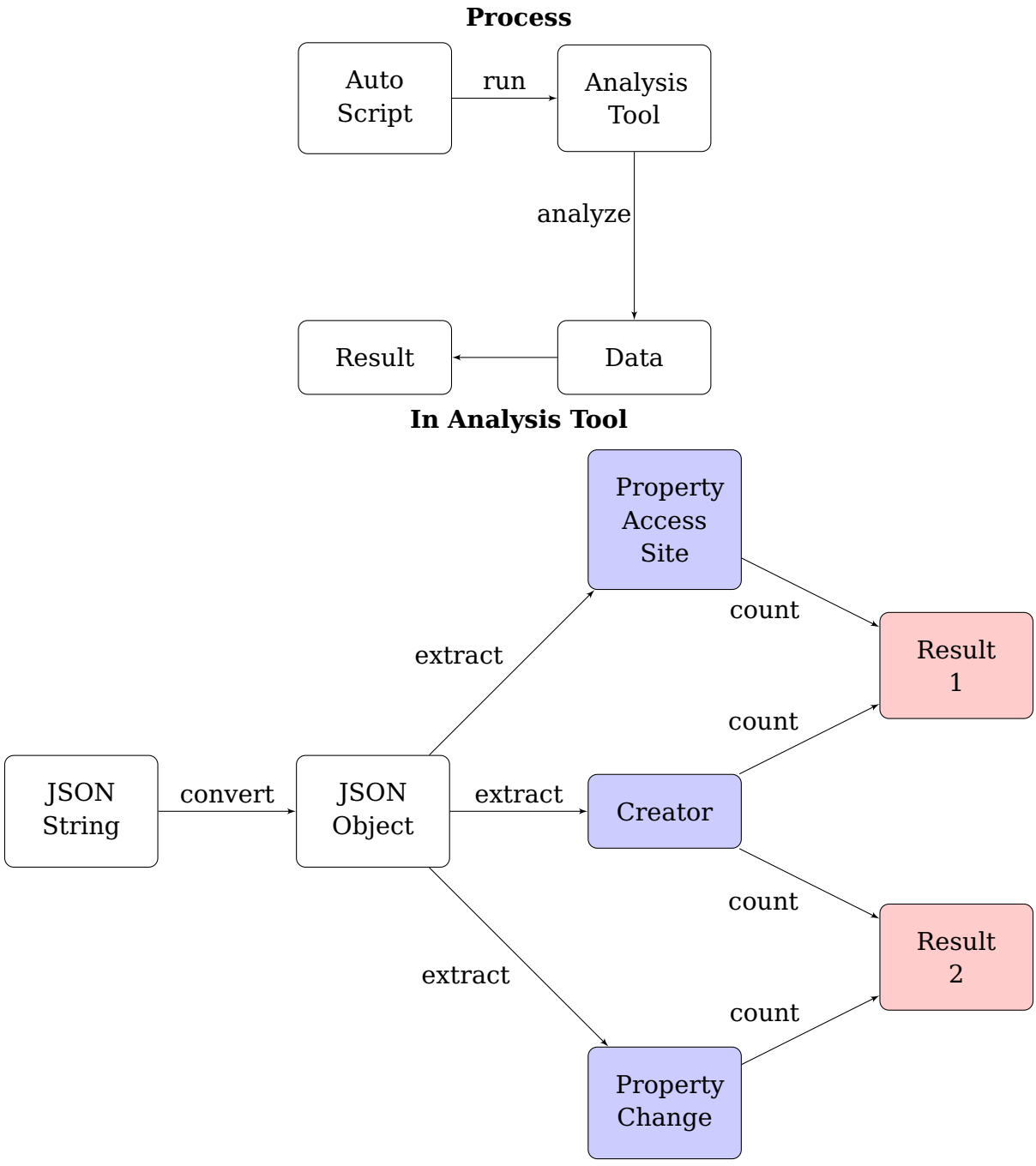

Figure 3.5: Detailed analysis process.

Table 3.2: Recorded data about access site.

creator	variableKeys	accessor	property	URI	lineno	start	end
foo	[x,y]	foo	x	demo.js	3	8	16
Object	[a,b]	foo	a	demo.js	4	22	30

3.4 Recorded Data

As mentioned, we tried to record data from the most popular 10,000 web sites as indexed by Alexa, in April, 2017. From these web sites, DotInstr successfully recorded about 80% of them, which produced 390 GB of JSON data as log files. Since the number of web sites is obviously far too large to visit manually, we resorted to a simple automation tool to launch a browser, in this work Firefox, with each URL from the Alexa list.

DotInstr opens each web site in Firefox with a configuration file to assure that the browser runs in a new session without cached data, to avoid the affection of history. Then, it waits one minute for the home page to load data and record the execution of JavaScript.

As our system is based on dynamic analysis, it is likely to miss some possible behaviors. Therefore, unseen events may render access sites or creators more dynamic than we observed.

In total, it takes us approximately seven days to finish recording all web sites, and several hours for one round of analysis.

To explain how our analysis tool works on recorded data, we take the two objects created in function `foo` as examples. One is initialized directly (i.e. `var m = {}`), and the other is created by the `Object` constructor (i.e. `var n = new Object()`). The two objects could generate the data shown in Table 3.2.

According to the previous chapters, the creator for the first object is the outer function(`foo` in this case) while the creator for the second object is its corresponding constructor, `Object`. Thus the creator on the first row of Table 3.2 is `foo`, and the creator on the second row is `Object`. Note that the `Object` constructor itself is rarely used; it is far more common to define constructors for the types used in a given program. DotInstr also records other data like the location information on the last four columns: `URI`, `lineno`, `start`, and `end`. `URI` is the full path of the source-code file. `lineno` indicates the line number of the access site. `start` and `end` are the column numbers indicating the start and end positions in source code of the access

Table 3.3: Recorded data about creators. Figure 3.6 generates the data.

creator	propertyType	isWrite	isNewProperty	newType	URI
foo	number	true	false	number	demo.js
Object	null	true	true	number	demo.js
Object	number	true	false	string	demo.js

site. The analysis tool uses such data fields to distinguish between different property access site and count creators on each site.

Besides the creator and its corresponding data fields, DotInstr recorded modifications of types on properties as well, shown in Table 3.3.

The code shown in Figure 3.6 can generate the three columns of data in Table 3.3. In Figure 3.6, the function `foo` calls the function `bar` with an object `obj1`. The creator of `obj1` is `foo`. When the sample code goes to line 2, which is `obj.val = 10`, DotInstr records data about the property access site. The data is the first column in Table 3.3. At line 10 in Figure 3.6, the object `obj2` is created by the `Object` constructor, which is the creator of `obj2`. At line 11, DotInstr records data about the property access site, which is the second column in Table 3.3. Then at line 12, the type of `val` changes from `number` to `string`, which generates the data at the third column in Table 3.3.

```
1           function bar(obj){
2               obj.val = 10;
3           }
4
5           function foo(){
6               var obj1 = {val:1};
7               bar(obj1);
8           }
9
10          var obj2 = new Object();
11          obj2.val = 2;
12          obj2.val = "20";
```

Figure 3.6: Sample code that generates the data in Table 3.3.

By the name of creator and the URI, the analysis tool distinguishes different creators. With the other four fields in Table 3.3, the analysis tool can verify whether there are modifications of types in objects created by these creators. Assuming that the object is accessed outside of its creator, if `isWrite` is `true` and `propertyType`

does not equal to `newType`, then it means that the type of property inside this object changed in a writing operation. And if `isNewProperty` is true, which means the code adds a non-existent property in the object, we also take the type of property as changed. It is not possible to add a new property after the object has been created in Java.

To summarize, the analysis tool use the recorded data for the two purposes:

(i) Counting the number of creators for every property access site with location information like the line number in JavaScript source-code. It provides analysis tool with the information about:

  (a) Names of creators.

  (b) Distinguishable location information for property access site. Combining the URI of the file with the line number, start and end position in the source code, the analysis tool can identify each property access site.

  Therefore, the analysis tool using both of them is able to generate results about whether there are multiple creators on property access-sites.

(ii) Checking changes about types of objects' properties on every property access site. The logs include the following data:

  (a) Identities of creators. By name of URI and the name of creator itself, it is able to find identical creators.

  (b) The type of the property before and after writing operation. The word "type" here is checked by `typeof` or the creator if the property is an object.

  (c) Whether there is a writing operation on existing property.

  (d) Name of the creator and accessing function. If the two items equal, the property access site is inside of the creator. The analysis tool does not count changes inside the creator.

  (e) Whether the accessed property is a new property that does not exist before the operation.

  The analysis tool could find changes occurring on objects created by a creator through the data.

3.5 Limitations

DotInstr assumes that developers may use many kinds of dot notations in their programs, such as `x.y` and `console.log()`. And it tries to convert these code to our customized code without knowing whether there are other usage unexpected. As a result, there are issues when working on some web sites with unforeseen code.

We cannot open about 2,000 web sites in Alexa's list with DotInstr. The browser feeds back nothing except an error message of "forbid to access". We have not yet figured out the cause for such problems. Thus there are no log files for these web sites.

Besides, in the remaining web sites, DotInstr failed to modify the AST nodes for recording logs in some special cases. Then it may affect the final result on web sites that cause this problem. We have resolved some issues on it. Taking `console.log` as an example, DotInstr relies on `bind` to `Function.bind` the context of `log` to `console` in its own functions instead of the original context. If there are user defined `bind` functions, DotInstr may run into errors since it still call the name `bind`. The solution is to define a variable pointing to `Function.bind` directly and call this variable as a function.

There are still other cases that may cause issues in DotInstr. For example all methods on the browser side of DotInstr may be re-defined by web-sites programmers and thus cause problems in DotInstr. We can fix most of the issues if we figure out the causes like the previous example of `bind`.

There are two kinds of property accessors: dot notation and bracket notation. DotInstr only focuses on dot nation like `object.property` but not on bracket notation like `object['property']`.

Other aspects of future works include:

(i) Since DotInstr records logs during execution of JavaScript code, it causes a slowdown. For the home page of the web site like "amazon.com", it takes about 30 seconds to finish loading, which is slower than normal. For recording logs on all web sites, we set the waiting time to 60 seconds to make sure that the browser can load the home page successfully. As a result, the whole recording process took about one week. We may focus on improving the efficiency of recording process in DotInstr itself in the future.

(ii) Checking changes of types is based on the granularity of creator. The recorded data and analysis tool can distinguish among the various creators, but cannot

separate each object. For example, if one creator create two objects and the analysis tool find the type of a property changed, the analysis tool cannot determine in which object this property was changed. DotInstr did not record the name of objects. As a result, we cannot count the number of objects that have type changes. Although the result is aimed to show information of the creator, it lacks the ability to extend for showing more detailed information on objects.

(iii) For changes on properties of objects, DotInstr currently does not check whether there is removal of properties. For example, if a property is added then removed outside of the creator, how shall we deal with this property and the corresponding object? Shall we mark the creator as a stable one since modification has been removed? These questions are yet to answer.

To sum up, the most important issue is that DotInstr fails to access or work on some of the web sites. It leads to the result that more than 20% web sites fail to generate log files.

Once we fix issue, DotInstr could generate more representative results on both the number of creators and type changes in objects.

Chapter 4

Observations

This chapter describes the results of the study. To discover the existence of class-like behaviour, we underline the number of creators on each property access site and type changes from several perspectives. Data used in this chapter were recorded in April, 2017.

4.1 Number of Creators

In typed languages, types are typically annotated at the level of variables, except for languages employing automatic inference of the data type. DotInstr focuses on the number of creators at a slightly finer granularity, per access site. This allows for patterns of variable reuse that statically-typed languages do not support.

Counting creators per method is flawed as there might be multiple property access sites in one method. That is to say the number of creators in each method depends on the number of property access site. Therefore, we count creators per property access site.

The first metric is the number of creators for each property access site. We divide property access site into four categories depending on the number of creators of the site: single creator, two creators, three creators, and a single category for many creators, as the case of more than three creators is rare. The numbers in Table 4.1 shows the number of access site with single creator is the largest. The statistics conforms to our expectations with respect to the prediction of object creators.

Table 4.1: Almost all property access sites have a single creator.

Category of Creators	Occurrence	Percentage
Single Creator	15,578,611	99.70%
Two Creators	42,370	0.27%
Three Creators	2,540	0.016%
Many Creators	2,341	0.014%

As we can see, through the analysis of data collected by opening the home page of each web site, more than 99% of property access sites access properties from objects with only one creator. In the high-level view, what *does* happen is that JavaScript applications appear compatible with types, based on the number of creators on property access sites.

In addition to the number of creators in all tested web sites, we investigate the distribution of creator numbers in individual web sites. As there may be some special web sites including more or fewer access sites than others, the analysis for each log file corresponding to individual web site is necessary to avoid an overwhelming situation caused by the special web sites. To this end, we divided web sites into two categories:

(i) Single creator: all object property access sites in the web site have one creator.

(ii) Multiple creators: in this kind of web site, there are object property access sites containing more than one creator.

Table 4.2 shows the classification result of Alexa's top web sites. As mentioned in Section 3.5, since DotInstr cannot work on some of the web sites, while some others offered no logs, i.e., empty in log, the total number of single and multiple creator web sites does not equal to 10,000. The proportion of empty web site is only 1%.

Table 4.2: Numbers of single and multiple creators web sites are both substantial.

Web Site	Occurrence	Percentage
single creator	4,608	59%
multiple creators	3,113	40%
empty	32	1%

The percentage that single creator web sites take denotes that lots of JavaScript applications treat object creations similar to Java: each object has only one creator (or type).

In Table 4.2, we found that there are 3,113 web sites that contain access sites with multiple creators, which indicate that many JavaScript applications do not create objects with the notion of types. We did advanced research and provided further data to prove that most access sites in these web sites have only one creator.

To show more detailed information, Table 4.3 displays distributions of property access sites with different numbers of creators among these 3,113 web sites. Table 4.4 lists several examples of unusual behaviours from these web sites. Most of the property access sites still contain only one creator. The proportion of single creator in these selected web sites is greater than 98%, which is similar to the number in Table 4.1. We can see that although the number 2,899 on the second row of Table 4.3 is large, which denotes the occurrences of web sites containing property access site with two creators is large, the number of property access site with two creators as shown in Table 4.4 is quite small. Three and more creators are even more rare than two creators. Property access sites distribute similarly in Table 4.1 and Table 4.4. And among all the log files, we did not find any extreme web site that the single-creator access site is not the majority.

As a result, it is reasonable to state that the object itself is mainly created by one particular type, regardless of whether the object is in a multiple-creator web site. So in the following example:

```
function increment(x){
    x.y+=1;
}
```

if it is guaranteed that the argument x is always from creator,

```
function XC(){
    return {y:1}
}
```

then the JavaScript compiler could compile it with a type, which saves time by avoiding looking up the object and its properties, as the following example indicates.

```
function increment(XC x){
    (XC)x.y+=1;
}
```

Based on the analysis result in this section, we conclude that the notion of Java-like types exist and thus we can employ types in JavaScript.

Table 4.3: Most web sites contain single and two creators access sites.

Number of Creators	Occurrence in Web Sites	Percent
1	3,113	100%
2	2,899	93.2%
3	761	24.4%
4	346	11.1%
5	280	9.00%

Table 4.4: Access site with single-creator is the majority.

Web Site	Number of Creators				Frequency of Single Creator
	1	2	3	4	
oriflame.com	2,930	34	2	NA	98.7%
rae.es	2,357	1	4	0	99.7%
sony.jp	3,837	1	4	0	99.9%
ebay-kleinanzeigen.de	5,020	6	0	4	99.8%
money.pl	3,697	9	10	0	99.5%
bmo.com	6,755	34	11	1	99.3%
walmart.ca	6,984	76	5	0	98.9%
webex.com	4,452	76	2	3	98.2%
gumtree.com.au	3,886	7	1	1	99.8%

4.2 Category of Creators in Single and Multiple Access Sites

The data about number of creators on each access site can be inverted for determining whether particular creators are more prone to be used in property access site with multiple creators. Creators on property access site with multiple creators limit the application of types, since objects of property access site with multiple creators have more than one creator by nature and are not in any kind of types. In other words, if we treat a specific creator as a type but find this creator on the property access site with multiple creators, the notation of type fails on that access site. For example, the creators `Date` and `Number` are at a property access site with more than one creator, we cannot set the type of objects on this site as either of them, since the object can be an instance of either `Date` or `Number`.

To answer the question about whether we can employ creators as the types of

objects, we divide the creators into two types: ones that appear only in access sites with single creator, and ones that appear in access sites with multiple creators. Since the scope of the creator "global field"; i.e., the creator for objects created at the global scope, is much larger than any other ones, the analysis tool removes this creator from the result.

An example follows to show the two kinds of creators. The creator `Person` appears in property access site with one creator. And other two creators, `Car` and `Bicycle`, appear in site with more than one creator.

```
function Person(firstName,lastName){
    this.firstName = firstName;
    this.lastName = lastName;
}

function Car(price){
    this.price = price;
}

function Bicycle(price){
    this.price = price;
}


var p1 = new Person("P1","Robot");
// this is a property access site with single creator
// the creator 'Person' appears here
console.log(p1.firstName);

function foo(obj){
    // this is a property access site with multiple creators
    // creators 'Car' and 'Bicycle' are used here but 'Person' is not
    console.log(obj.price);
}

var car = new Car(22000);
foo(car);
var bike = new Bicycle(1600);
foo(bike);
```

From our analysis result, there are in total 48,758 creators that only appear in single-creator access sites, compared to 11,650 creators in multiple-creator access sites. Although the first kind of creators form a majority with the percentage 81%, it is incomparable with the result in Table 4.1 that access sites with single creator

takes the vast majority with the percentage of 99%. We will need more detailed data to examine the remaining 19% creators appears in multi-creator site.

As JavaScript built-in functions cannot be modified, we expected them to behave different from user-defined functions. The analysis tool compares the name of creators with a standard built-in functions list, as provided by Mozilla's developer web site [2]. Thus we can distinguish between built-in functions and user-defined ones.

In the analysis result of built-in creators in Figure 4.1, if there are twenty creators used only in property access site with single creator in a web site, and one of them is a JavaScript built-in creator, then the rate is 1/20 thus 5%. This web site will be in the bin "0% - 10%" in this figure. "Bin" means the rate of JavaScript built-in creators to all creators on single or multiple creators access sites. We can read from the table of Figure 4.1, in the bin "0% - 10%", there are 1,626 such kind of web sites where the rate of built-in creators is similar to the previous example.

Figure 4.1 shows that web sites labeled by the rate of built-in creators to creators on single-creator access site are distributed more evenly than those labeled by the rate to multiple-creator access site, although there is no fixed pattern for this even distribution.

To our interest, most web sites labeled by the rate of built-in creators to multiple-creator access site are located at the bin "0% - 10%", which means that JavaScript built-in creators only account for a small number of polymorphic sites. Even if there is a small increase in the interval of 90% to 100%, most of the web sites have only one creator appearing in multiple-creator access site; that creator is the JavaScript built-in `Object` creator.

From the result, we found that most creators reaching multiple-creator access sites are user-defined. Besides, among the 437 web sites at the bin "90% - 100%" represented by the right-most red bar in Figure 4.1, 419 web sites contain the creator `Object`. Thus we expect objects created by `Object` to have more than one creator.

The analysis result indicates that objects created by user-defined functions and the `Object` constructor are prone to have more than one creator. Thus we can not set types for these objects. Besides, whether objects from multiple creators reach access sites is only one aspect of type-like behaviour. In statically typed languages, object fields typically cannot change their types. The other aspect relates to types and properties of objects.

Table and Corresponding Bar Diagram

Rate Bin	Built-In to Single Creators	Built-In to Multiple Creators
0% - 10%	1,626	6,569
10% - 20%	349	148
20% - 30%	681	246
30% - 40%	877	126
40% - 50%	581	14
50% - 60%	1157	155
60% - 70%	684	23
70% - 80%	379	2
80% - 90%	343	1
90% - 100%	957	437
Total Web Sites	7,753	7,753

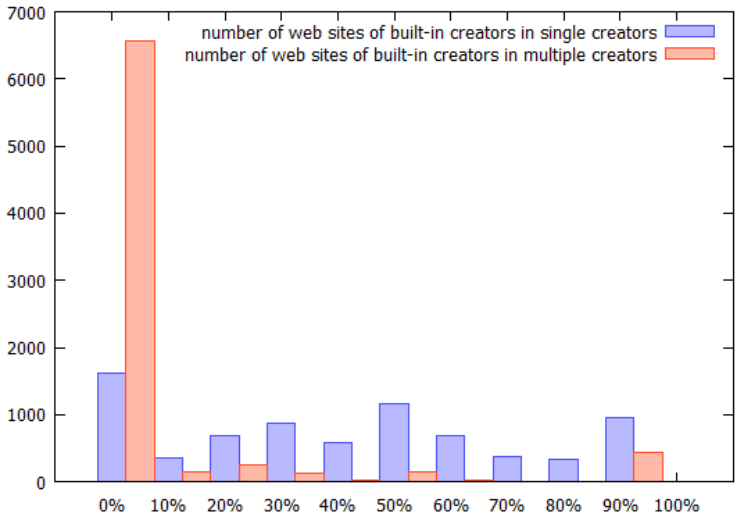

Figure 4.1: The rate of built-in creators to multiple-creator is low in most web sites.

4.3 Types of Properties in Objects

In the previous section, we explored the number of creators reaching each access site, as a way of understanding whether nominal types make sense for JavaScript. To learn more about JavaScript objects in real-world applications, we discuss types and existence of properties in this section.

As we know, types and existence of an object properties are fixed after creation in Java, while almost everything including both the type and existence of properties in JavaScript is dynamic. We need to determine whether there is modification of types on properties in objects and creations of new properties. If types of object properties in modern JavaScript web sites rarely change, combining with the result in Section 4.1, we could apply types and classes to JavaScript objects.

We divide creators into four types based on object properties:

(i) If there are no writing operations on object properties, we mark the creator of the object as “read only”.

(ii) If there are writing operations but only type-stable ones, which are writing operations without changing properties’ types, we mark this creator as “type stable”.

(iii) If there are writing operations that change the type of properties, we mark the creator as “dynamic”, to be more precise as “type dynamic”.

(iv) If there are creations of non-existent properties, we mark the creator as “fully dynamic”.

We can treat creators marked other than “dynamic” as Java classes in principle, as there are no changes of types during execution. The two types of “dynamic” creators are not Java-like.

Similar to Section 4.1, we focus on the result across all web sites at first. The analysis tool counts the number of these four kinds of creators as shown in Table 4.5. As we can see, although most creators belong to the “read only” and “type stable” categories, which are Java-like classes, there is a substantial proportion of “dynamic” creators. In following discussions, we use “stable” to describe “read only” and “type stable” creators, and “unstable” to describe both “dynamic” creators. Other than the 75% stable creators, there are 25% unstable ones. And comparing with “type

Table 4.5: “read only” creators are much more than others.

Type of Creator	Occurrence	Percentage
read only	52,197	69.0%
type stable	4,564	6.0%
type dynamic	4,394	5.8%
fully dynamic	14,509	19.2%

dynamic” creators, “fully dynamic” ones are the majority in unstable creators. The comparison indicates that creations of non-existent properties is the main cause for unstable creators.

For the question whether JavaScript code behaves similarly to statically-typed code, the fact that objects created from dynamic creators are not type-stable makes the answer more complex than the previous analysis of access site. Any notion of classes is less useful when there are changes of types or creations of new properties. Therefore, we need more detailed information about the dynamic creators.

To show results in a finer granularity, we classify the tested web sites more detailed into categories mentioned in section 3 based on the occurrences of read-only, type-stable, and dynamic creators.

(i) If there are only “read only” creators, we classify this web as a “all read only” one.

(ii) If there are “read only” creators and others, we classify this web site as a “partial read only” one.

(iii) If there are “type stable” creators but no “dynamic” ones, we classify this web site as an “all type stable” one.

(iv) If there are “type stable” creators and others, we classify this web as a “partial type stable” one.

(v) The rest are “dynamic” web sites.

Table 4.6 shows the result of number and percentage of web sites that belong to each of the categories. It should be noted that we discard the creator “global field” in all results of this section with the similar reason in Section 4.2. There are 623 web sites that have no creators other than “global field”, which is the “Unclassified”

web sites in Table 4.6. They account for about 8% of all recorded web sites. These web sites are meaningless for this analysis in the current analysis tool. So in this section, we do not classify and discuss them.

Table 4.6: Partial read-only web sites are the majority.

Category of Web Site	Number of Web Site	Percentage
All Read Only	1,269	16.37%
Partial Read Only	5,401	69.66%
All Type Stable	24	0.31%
Partial Type Stable	10	0.13%
Dynamic	426	5.49%
Unclassified	623	8.04%

From Table 4.6, we conclude that most web sites are “Partial Read Only”. “All Read Only” web sites account for about 16%. Since stable web sites, which are “All Read Only” and “All Type Stable” ones, do not account for most, it is hard to tell whether most creators are stable in a web site. But it is obvious for us to determine which kind of creators tend to be dynamic creators by their names. For example, if dynamic creators are most user-defined ones, we can employ classes for JavaScript built-in creators.

The “Partial Read Only” web sites, which account for 69.63%, attract our attention on what causes them to be “partial” but not “all”. If the reason is that unstable creators account for a substantial part, that means the improvement based on the application of classes may be limited. Alternatively, if unstable changes represent a minority, similar to result about property access site with multiple creators comparing to single creator, there is a good chance to improve the efficiency of JavaScript. More detailed explanation is in Section 4.4.

To approach the causes for “Partial Read Only” web sites from another perspective, we concern unstable writing operations, which include changing types of properties and creating non-existent properties. Table 4.7 lists our definitions of operations. Considering that stable web sites only account for a small portion, we expect that stable writing operations are not the majority in modern JavaScript applications. Therefore, we need to examine whether stable writing operations lead to “Partial Read Only”. The problem becomes simpler if stable writing operations make it “partial”. The discussion is in Section 4.5.

Table 4.7: Definitions of operations.

Definition of Writing	Description
unstable writing operation	Value and type of property changed during the writing operation.
stable writing operation	Only value of property changed.
creation of new property	A non-existent property is created in the object.

4.4 A Discussion of Creators on "Partial Read Only" Web Sites

Since "Partial Read Only" web sites account for about 70% of all, it is valuable for us to run a deep analysis about creators on such web sites.

At first, we updated the analysis tool to count dynamic creators among all types of creators in each web site. We expect to see whether there are more dynamic creators than others in a web site. Besides, since dynamic creators suggest very un-class-like behaviours, we need to explore the reason for the existence of dynamic creators.

We separate web sites according to the percentage of dynamic creators to all creators by the stride 10%. The result is in Figure 4.2. For example, if there are totally twenty creators in a web site, and five of them are dynamic creators, then the percentage of dynamic creator is 25%. This web site goes to the bin "20% - 30%". From the table, we conclude that in "Partial Read Only" web sites, there are a large number of dynamic changes. Most web sites are in the bin from 0% to 70%. In the bin "20% - 30%", it has the most web sites, 1446 ones. Thus, we argue that JavaScript creators exhibit highly dynamic types with a reasonable percentage. Even if most property access sites, which are more than 99%, have only one creator, types of properties in objects created by the creator change. Therefore, we cannot treat creators as classes.

However, there are still stable creators among them, and the portion is considerable. For example, the bin "0% - 10%" means other 90% to 100% creators belong to stable creators. If we can differentiate them based on some features of creators, there could still be room for improvement. Among web sites that contain dynamic creators, there is a possibility that all other stable creators belong to one category, like user-defined functions or JavaScript built-in functions. If objects created from

Among Partial Read Only Web Sites

Percentage of Dynamic Creators	Number of Web Sites	Percent of Web Site
0% - 10%	650	12.0%
10% - 20%	1,050	19.4%
20% - 30%	1,446	26.8%
30% - 40%	1,111	20.6%
40% - 50%	511	9.5%
50% - 60%	461	8.5%
60% - 70%	139	2.6%
70% - 80%	24	0.4%
80% - 90%	9	0.2%
90% - 100%	0	0%

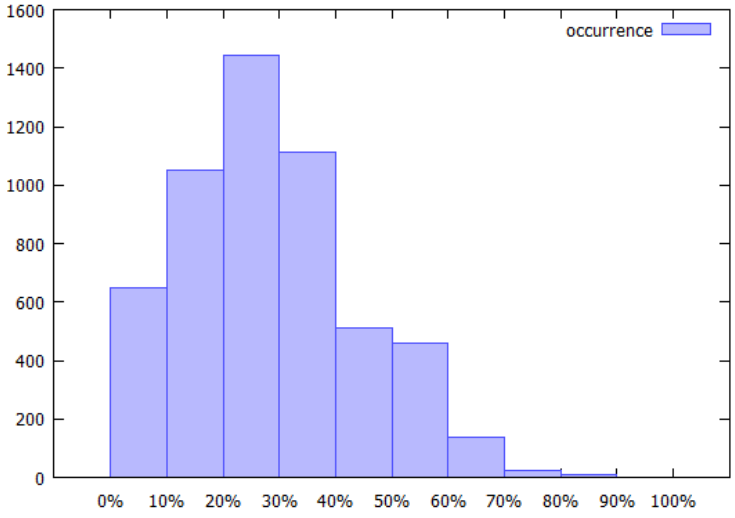

Figure 4.2: There exist substantial dynamic creators.

one particular category of creators rarely change during execution, we can apply make them work with the notion of “class”.

We now distinguish type stability between user-defined and built-in creators, expecting to find a pattern that only objects created from one kind of creator are prone to change during execution.

Figure 4.3 is the result of this further study. We separate web sites according to the percentage of JavaScript built-in creators to all dynamic creators by the stride 10%. There are two sub-bars at bin “0% - 10%” and “90% - 100%”. The first red sub-bar at “0% - 10%” shows special web sites that are at the critical point 0%. And the second green sub-bar at “90% - 100%” shows the web sites that contain the special built-in creator `Object`. We will present further discussions of the two cases. As shown, most objects created by JavaScript built-in functions are stable. Web sites ranging from 0% to 10% accounts most, which denotes that most web sites only contain small portions of dynamic built-in creators. Type instability in these web sites is most likely from user-defined creators. Based on this observation, we have a chance to treat built-in creators as classes.

Looking further into the range from 0% to 10%, 2,783 web sites among 2,866 are actually at the critical point 0%, where 0% indicates that a web site exists dynamic creators but none of them are built-in creators. That is to say, there are significant number of web sites contains 0% of JavaScript built-in creators among all dynamic creators. Further, the bin “90% - 100%” is an outlier, compared with that from 60% to 90%. After checking the 460 web sites in the bin “90% - 100%”, we found that all these web sites are actually at the point 100%, where 100% means dynamic creators in the web site are all JavaScript built-in creators.

Considering the two extreme cases with percentage 0% and 100%, we count the creators to seek the pattern in different built-in creators. For web sites at the point 100%, where all dynamic creators are JavaScript built-in ones, there are 360 out of 460 containing the creator `Object`. This is reasonable since the `Object` constructor does not relate to types. In this case, we expect `Object` to be used similarly to object literals. For web sites at the point 0%, ones having no unstable JavaScript built-in creators, there are only 463 of 2,783 instances containing the creator `Object`, accounting for about 16%.

Table 4.8 and Table 4.9 show that JavaScript built-in functions are more related to types except the `Object` constructor, as seen in the following examples.

(i) RegExp. The constructor creates an object for a regular expression. There is no need to add new properties or change properties’ types, if the object is

Among Web Sites Contain Dynamic Creators

Percentage of Built-In Creators	Number of Web Sites	Percentage of Web Sites
0% - 10%	2,866	53.0%
10% - 20%	291	5.4%
20% - 30%	494	9.1%
30% - 40%	361	6.7%
40% - 50%	99	1.8%
50% - 60%	649	12.0%
60% - 70%	145	2.7%
70% - 80%	29	0.5%
80% - 90%	12	0.2%
90% - 100%	460	8.5%

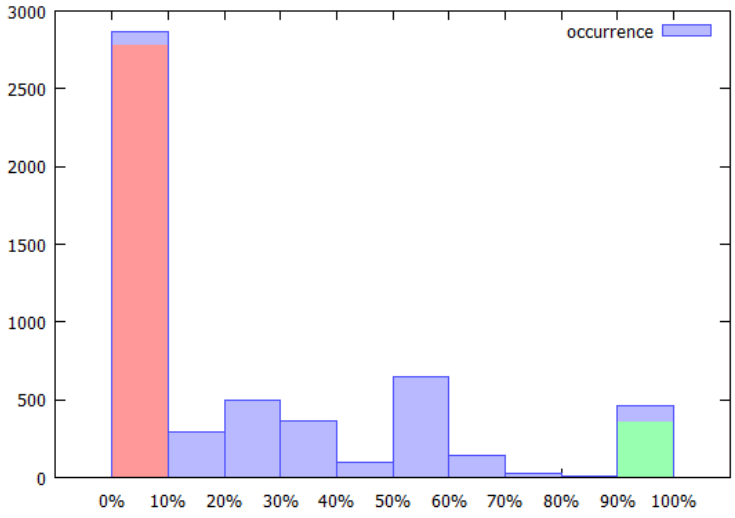

Figure 4.3: Built-in creators account for a small portion in dynamic ones.

Table 4.8: Three stable built-in creators most commonly used.

Among "0%" Web Sites	RegExp	Date	Array
Web Sites Using the Creator	1,747	1,510	249
Percentage	62.8%	54.3%	9.0%
Among "100%" Web Sites	RegExp	Date	Array
Web Sites Using the Creator	1	1	2
Percentage	0.2%	0.2%	0.4%

utilized as a regular expression. Normally, there are two arguments in the constructor.

```
new RegExp(text, flags);
```

The first argument is the regular expression itself, and the second one is flags for how the expression is to be interpreted. Changing types or adding new properties make no sense on objects created by `RegExp`. As shown in Table 4.8, there are 1,747 occurrences of `RegExp` as stable creators, while there is only one case using `RegExp` as a dynamic creator.

(ii) Date. The constructor is to generate an object representing a moment in time. Properties are time-related, such as years, months, days, etc, as precise as milliseconds. Even if programmers want to add more properties, the built-in date-related functions would not use them. As shown in Table 4.8, only one case employs `Date` as a dynamic creator.

(iii) Array. For objects created with `Array` constructor, they are obviously used as arrays. Operations are based on bracket notations and array operations, such as `arr[0]`, `arr.push(1)`, etc. In principle, arrays could carry any type and their types could change, but we find arrays are seldom unstable.

The above examples are the most common JavaScript built-in creators among the "0%" web sites. While the three creators are majorities in the web sites that contain 0% dynamic JavaScript built-in creators, they are the minorities among the "100%" web sites. On contrary, the `Object` creator is highly dynamic, which occurs in about 78% of all the web sites that contain 100% dynamic JavaScript built-in creators.

We could treat the three stable built-in creators as Java-like classes in current code. In future analysis, we may find more characterized creators for improvement.

Table 4.9: The creator `Object` is unstable.

Among “0%” Web Sites	Object
Occurrences	463
Percentage	16.6%
Among “100%” Web Sites	Object
Web Sites Using the Creator	360
Percentage	78.3%

In future work, it would be necessary to verify whether the amount of objects created from stable JavaScript built-in creators account for enough number to be worth focusing on. Although most JavaScript built-in creators themselves are stable, if there is a small number of such objects created and used in modern JavaScript applications by developers, then it would be less valuable for us to focus on them. However, as described in Section 3.5, DotInstr is not currently able to distinguish different individual objects among property access sites. Such a distinction is necessary for us to get a greater representativeness and accuracy.

4.5 Statistics of Unstable Writing Operations

The analysis tool also checks the total number of writing operations that lead to changing types of property for each dynamic creator. The rule is strict for labeling the creator as a stable one: one change of type or new property could lead to the creator being noted as dynamic.

If unstable writing operations are sufficiently rare, we could focus on the following perspectives:

(i) When do unstable operations take place? If they arise at the very beginning after creation, the JavaScript engine could employ a function to re-build the creator based on the change, so that code after the change can use the new creator in a stable way. We then could consider the new creator as “read only” or “type stable”.

(ii) Are there unstable writing operations on most objects created from the creator, or only a few? Once the log files and analysis tools are capable of checking and distinguishing individual objects from the same creator, there is a chance for

us to answer this question. If there are unstable writing operations on only a few objects, but causing the creator to be an unstable one, it may be possible for us to characterize the problematic objects and separate them from stable ones.

For this purpose, we analyze the total number of unstable writing operations on each web site containing dynamic creators, compared to stable ones. We calculate the rate of unstable writing operations to stable ones. Stable writing operations lead to "Partial Read Only" web sites as well as unstable ones. In Figure 4.4, we examine which kind of operations is the main cause of "Partial Read Only" web sites. A high rate means unstable writing operations is the main cause in a web site. From the calculation result, unstable writing operations are the majority, as the range from 80% to 100% accounts for more than half of web sites, which means such many web sites have the rate higher than 80%. The figure reveals that in most web sites containing dynamic creators, writing operations on these creators are most likely unstable.

4.6 Case Analysis

4.6.1 Analysis of Unusual Web Sites

In this section, we analyze and characterize examples of web sites belonging to the following four unusual or surprising categories.

(i) All property access sites are single-creator, and most creators are stable, i.e. "read only" or "type stable". These web sites show highly type-like behaviour. Examples selected are: google.com, google.si, irs.gov, and cda.pl.

(ii) Most creators are stable, but there exist several cases of multiple-creator property access sites. The code in this kind of web sites would be easy to rewrite with classes, but difficult to annotate with types. Examples selected are: nyc.gov, wrestlinginc.com, tickld.com, and hizliresim.com.

(iii) There exist cases of multiple-creator access sites, and most creators are "dynamic". These web sites are highly dynamic. Examples selected are: w3.org, ilsole24ore.com and launchpad.net.

Among Web Sites Contain Dynamic Creators

Rate Bin	Number of Web Sites	Percentage
0% - 10%	143	2.6%
10% - 20%	147	2.7%
20% - 30%	171	3.2%
30% - 40%	251	4.6%
40% - 50%	326	6.0%
50% - 60%	408	7.5%
60% - 70%	581	10.7%
70% - 80%	500	9.2%
80% - 90%	689	12.7%
90% - 100%	2,190	40.5%

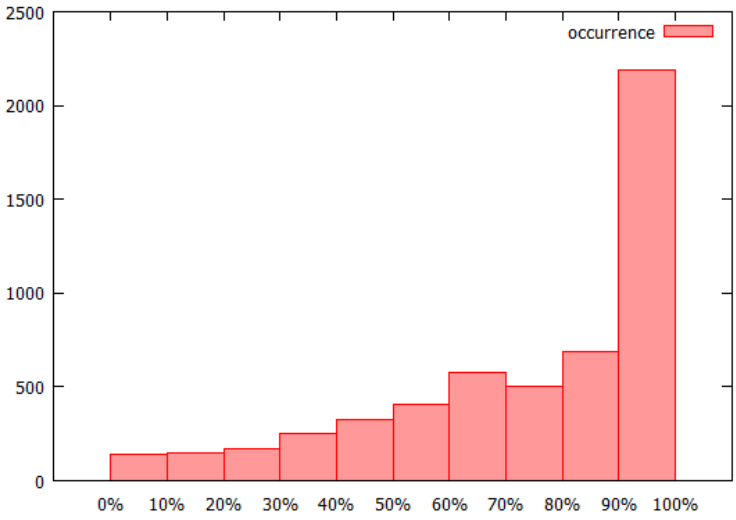

Figure 4.4: Rate of unstable writing are much more than stable ones.

(iv) The case where most access sites have one creator, but most creators are "dynamic". These web sites do not correspond well to static types, as their annotations would be fairly nominal, but they could not be rewritten into classes. Examples selected are: cameraprive.com, searchmgr.com, ucdavis.edu, and vagas.com.br.

From category i, in cda.pl, there is one dynamic creator and most writing operations are unstable, which makes it hard to decide whether it is possible to apply types. After checking the data behind Table 4.9 for this web site, we find that the only dynamic creator is the `Object` constructor, so others are stable.

Web sites belonging to category ii are expected to have more complex combinations than that in category i. There is a high incidence of unstable writing operations on the four web sites. And for nyc.gov, dynamic JavaScript built-in creators account for half of all. So it is impossible to apply the improvement on all creators in these web sites.

From the web sites in category iii and iv, we find that some of the dynamic creators are from JavaScript built-in functions but others are not. For example, none of the creators in ilsole24ore.com are from JavaScript built-in functions. But on launchpad.net, they are all from JavaScript built-in ones. In this case, DotInstr could not decide whether it is possible to guarantee the improvement on them because there are no further data to help for making decisions.

4.6.2 Analysis of Source Code from Three Examples

Additionally, we analyzed complicated web sites in further detail, with raw results shown in Appendix A.1. We selected three web sites due to their special cases of creators and type changes.

In the three selected web sites with more detailed data, we counted and compared the number of creators in the two fields:

(i) Under the cases of single-creator access site, two-creators access site, etc., we extracted the name of creators and their occurrences.

(ii) For the four kinds of creators, we examined their names and numbers.

As we can infer from the raw data, most creators under the cases of multiple-creator access sites and dynamic creators are not in the list of JavaScript built-in functions. That means these creators are from user-defined ones. After comparing names in the collection of single-creator access sites with that in the collection of stable ones, we found that the two groups are not related to each other. The creator that is found in the collection of single-creator access sites may not be found in stable creators, and vice-versa. These two notions of type stability seem unrelated to each other.

To understand why these surprising cases of multiple-creator access sites and dynamic creators arise, we also compared the source code of web sites and their corresponding logs. We found several interesting examples in this process.

Analysis of abc.net.au

In abc.net.au, there are 39 "read only" creators, 14 "type stable" ones and 14 "dynamic" ones. The case is special since there exist several creators in each type. We count the creators in abc.net.au and find that the three creators in Table 4.8 are all stable. And the `Object` creator is dynamic in abc.net.au, similar to the result in Table 4.9. The raw results also justify the argument that the two creators `Date` and `RegExp` only appear in cases of single-creator access sites and stable creators.

Besides, there are three code examples for "dynamic" creators and one for the multiple-creator access site.

First, we found that in a creator function named "o", it defines an object with many simple properties: getters and setters, `addListener` and `removeListener`, etc. However, while returning this object, in a function property of this object, it redefines the `status` property that already exists.

```
function o(){
    var ret = {};
    ret.id = id;
    ret.name = o;
    ret.status = {};
    // define properties of ret
    return ret.getters() = function(){},
    // define function properties of ret
    ret.updateStatus = function(info){
        ret.status = {
            // update status based on information extracted from 'info'
            id:info.id,
            start:info.start,
```

```
            end:info.end
        }
    };
}
```

The function `updateStatus` is like an API for others to change the `status` property of this object outside of its creator, but in a controlled way.

On the one hand, from this example, we can see that if an object property is treated as a collection of data, it is quite possible for the property itself, and not just the data in it, to be updated.

On the other hand, this example suggests that our notion of creator may need some refinement. While the two objects reaching `ret.status` do indeed arise from different creators, both are within a same function.

The second special usage is: while a property `initlized` has already been defined by its creator, another function defines a similar property name, i.e. `init`, in the same object rather than using `initlized`.

Another example shows the way JavaScript programmers update data fields with dynamic features. We use a simplified example to show the usage. The first approach of updating follows.

```
function creatorF(){
    return {count:{content: val1, static: val2}};
}

// obj is created by function creatorF
function use(obj){
    // do something and then update count
    obj.count = {content: val3, static: val4, id: val5};
    // count is now from another different creator
}
```

The object `count` is updated by assigning a new instance instead of setting new values to its properties one by one, which is the second approach.

```
// obj is created by function creatorF
function use(obj){
    // do something and then update count
    obj.count.content = val3;
    obj.count.static = val4;
    obj.count.id = val5;
    // count is still created by function creatorF
}
```

In the first approach, the property `count` in object `obj` is assigned a new value with creator `use`, instead of `creatorF`. As a result, `obj` will be marked as “dynamic”. In the second way, even if `count` is made “dynamic”, the instance `obj` is still stable. However, the first approach is easy to use, an thus a more common way for updating data fields in the object.

For the code example of multiple-creator access site on this web site, there is only one cause: a function taking an object as its parameter is called everywhere in many other functions. The code example follows.

```
function foo(obj){
    //access properties
    obj.p1 = v3;
}

function caller1(){
    foo({p1:v1, p2:v2});
}

function caller2(){
    foo({p1:v1, p2:v2});
}

function caller3(){
    foo({p1:v1, p2:v2});
}

caller1();
caller2();
caller3();
```

Each time `foo` is called in other functions like `caller1`, a different creator reaches the property access site. There will be three different creators `caller1`, `caller2`, and `caller3` in function `foo` reaching its property access site.

Besides the mentioned web site, there are two other web sites to be analyzed with surprising cases as well: schwab.com and timeout.com. After analysis, we found that there are reasonable causes for cases of multiple creators and “dynamic” creators in each. Among all causes, there exist patterns that lead to “dynamic” creators.

Analysis of schwab.com

In the web site schwab.com, the first example shows that functions are called like `object.handle(arguments)`. There are many examples of `object.handle`, where `object` is created by different functions and `handle` is an inner function to deal with input arguments. The code follows.

```
function c1{
    function c1h(input1){}
    return {handle:c1h}
}

function c2{
    function c2h(input2){}
    return {handle:c2h}
}

//obj is from different creators
function use(obj){
    // do some jobs first
    obj.handle(input);
}
```

Another special case shows that when a function checks whether a particular property exists, as well as its equality to a value:

```
function check(obj){
    return obj.pro1 && obj.pro1 === value1;
}
```

there is a high possibility that `obj` is from different creators considering the purpose of this function and the lack of a type system in JavaScript. So access sites inside the function `check` will be marked as from multiple creators.

For "dynamic" creators in this web site, there are five main causes listed below:

(i) A function adds new fields and functions to its object argument.

(ii) Outside of the object creator, there is a checking and adding operation of non-existent properties like `obj.p = obj.p || {};`.

(iii) Updating existing object-fields like `obj.p = {k1:v1,k2:v2}` instead of `obj.p.k-1=v1, obj.p.k2=v2`. From this kind of usage, the creator of `p` changed and

thus caused the creator of `obj` to be a dynamic one, according to our definition. However, if the creator changes but the structure is the same, we could still treat these creators as the same type in structural typing system.

(iv) Other unusual additions of fields, like `var obj = creator(); obj.field = creator;` outside of the creator function.

(v) Objects are created by `Object` constructor and added new fields named like `data`.

Analysis of timeout.com

In the last analyzed web site timeout.com, there is only one reason for cases of multiple creators: for-in based reflection. The code follows.

```
for(var i in objLoop){
    if(objLoop.hasOwnProperty(i)){
        var obj = objLoop[i];
        // an access site of properties in obj
        console.log(obj.val);
    }
}
```

For "dynamic" creators in this web site, there are three causes. The first is similar to that in abc.net.au: using fields named with `init`, `initialized` and `initializing` as if they already exist. Second is that the object creator opens an interface to others to change its properties, so that functions other than the creator may change its fields, and then make it "dynamic". The last cause we classify as truly surprising: some objects simply change outside of their creators with no particular pattern.

After the analysis of all three web sites, we achieved the following conclusions.

(i) For cases of multiple-creator, there is one site in the code, e.g. an argument of the function is an object that created by different creators, from which multiple creators originate. After that site, any property access site is marked as with multiple creators.

(ii) For "dynamic" creators, the common patterns are: updating object fields with an object instead of these properties one by one, and using `init`-like properties as if they already exist. Other causes appeared patternless.

4.7 Web Sites that Break DotInstr

DotInstr failed to work on some web sites. The cause for most failures is that DotInstr cannot preserve the original meaning of the code after modifications. Since DotInstr modifies the structure and content of the AST node, there are several cases where the modification breaks the structure of AST or makes modified JavaScript code invalid.

After catching exceptions about serializing AST back to JavaScript code, DotInstr restores the code to its original version without modification. DotInstr may fail to restore the original version, which causes the second kind of failure. Then the web sites may stop loading and produce no log files whatsoever. As mentioned in the previous chapter, some of web sites do not provide accesses through DotInstr although they can be accessed directly. Error messages show that the access is forbidden when accessing with DotInstr. As a result, we finally got log files of more than 7,700 web sites, but not for all 10,000 ones.

4.8 Global Fields

When counting modifications on property types in objects, DotInstr discards the creator “global field”, which is the global scope under each JavaScript source-code file. Since the number of changes depends on the size of such field, DotInstr currently only focuses on creators other than “global field”, as global objects are unrelated. However, there are web sites in which all creators are “global field”. For such web sites, it is not possible to get useful information from them.

When DotInstr and the analysis tool can distinguish among individual objects, it would be possible for us to include these global obejcts in the result, by separating unrelated objects under the creator “global field”.

Chapter 5

Conclusion

The thesis provides a study and analysis to determine whether JavaScript code exhibits behaviour similar to classes in Java and other statically-typed languages, by information on property access sites and types of creators. Our study is based on data collected from more than 7,700 web sites on the Alexa top list. We built the tool used in this study on JSBench and named it DotInstr. Through DotInstr, we gathered data about object creations and property access sites during JavaScript execution. DotInstr saved data on the local disk in the format of JSON. We provided advice about whether it is possible to utilize "types" for performance improvement and other enhancements to dynamic code. We also measured how much improvement techniques like inline caching is expected to bring. After comparing class-like behaviour of objects in JavaScript applications with Java, the study achieved the result that object behaviour in JavaScript applications tend to be Java-like. After analyses on several different aspects, we got following results:

(i) We observed that most property access sites, more than 99%, access properties from one creator. The case is still similar in each web site that most property access sites are with one creator. While 40% of web sites contain multiple creators on property access sites, the total number of such access sites is small. Property access sites that have multiple creators account for less than 1%.

(ii) While most property access sites are with one creator, types of properties in objects changed a lot through writing operations. For all creators other than the global scope, there exist 25% "dynamic" creators, which means objects created by such creators do not follow the notion of classes as types and existences of properties in such objects change during execution.

(iii) Compared to unstable creators, we found that objects created by JavaScript built-in creators are more stable, except the creator `Object`. These built-in creators such as `Date` and `RegExp` could be treated as classes during execution.

We started this work expecting that types could be employed for performance improvements in modern JavaScript applications. However, the results suggest that this application is limited, and such improvements could not be employed directly without further condition checking. JavaScript programmers do use dynamic features, and thus restricting dynamism would break their original code. These dynamic aspects bring conveniences for JavaScript programmers.

In summary, the percentage of single-creator access sites cannot guarantee that modern JavaScript applications behave as if they were typed. In our current analysis result, only some of JavaScript built-in creators have the potential to be used as classes. JavaScript's dynamic nature is used to full effect in real code.

References

[1] Inline caching. https://en.wikipedia.org/wiki/Inline_caching. Accessed: 2017-06-17.

[2] JavaScript standard built-in objects. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects. Accessed: 2017-04-28.

[3] Typescript. https://www.typescriptlang.org/. Accessed: 2017-06-19.

[4] Sharath Gude, Munawar Hafiz, and Allen Wirfs-Brock. Javascript: The used parts. In *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*, pages 466–475. IEEE, 2014.

[5] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[6] 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*, pages 736–747. ACM, 2012.

[7] Frolin Ocariza, Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. An empirical study of client-side JavaScript bugs. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 55–64. IEEE, 2013.

[8] Michael Pradel and Koushik Sen. The good, the bad, and the ugly: An empirical study of implicit type conversions in JavaScript. In *LIPIcs-Leibniz International*

Proceedings in Informatics, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[9] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G Zorn. Jsmeter: Comparing the behavior of JavaScript benchmarks with real web applications. *WebApps*, 10:3–3, 2010.

[10] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated construction of JavaScript benchmarks. In *ACM SIGPLAN Notices*, volume 46, pages 677–694. ACM, 2011.

[11] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498. ACM, 2013.

[12] Leonardo Humberto Silva, Miguel Ramos, Marco Tulio Valente, Alexandre Bergel, and Nicolas Anquetil. Does JavaScript software embrace classes? In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 73–82. IEEE, 2015.

APPENDICES

Appendix A

JSON Data Extracted for Case Analysis

A.1 A Detailed JSON Data

+

abc.net.au:


```
{
   "isSingle":false,
   "isEmpty":false,
   "webSiteType":1,
   "creatorsNumber":{
      "16":1,
      "1":659,
      "17":9,
      "2":10,
      "3":1,
      "6":1,
      "7":4,
      "10":1,
      "14":3
   },
   "readOnly":41,
```

```
   "typeStable":14,
   "randomChange":12,
   "numberofCreOcc":{
      "16":{
         "b":1,
         "C":1,
         "D":1,
         "G":1,
         "H":1,
         "h":1,
         "J":1,
         "N":1,
         "O":1,
         "Q":1,
         "q":1,
         "v":1,
         "w":1,
         "x":1,
         "z":1,
         "_":1
      },
      "1":{
         "CustomEvent":2,
         "B":1,
         "MutationObserver":1,
         "G":4,
         "H":58,
         "O":3,
         "Q":8,
         "Pc":3,
         "ob":7,
         "HTMLImageElement":1,
         "_":17,
         "d":5,
         "RegExp":11,
         "g":5,
         "i":78,
         "NolTracker":25,
```

```
         "k":4,
         "m":8,
         "n":95,
         "Date":13,
         "tb":5,
         "o":127,
         "Array":2,
         "Promise":20,
         "pa":7,
         "pb":23,
         "r":5,
         "s":43,
         "u":9,
         "v":5,
         "w":7,
         "Object":55,
         "z":2
      },
      "17":{
         "b":9,
         "C":9,
         "D":9,
         "G":9,
         "H":9,
         "h":9,
         "J":9,
         "N":9,
         "O":9,
         "Q":9,
         "q":9,
         "r":9,
         "v":9,
         "w":9,
         "x":9,
         "z":9,
         "_":9
      },
      "2":{
```

```
            "Q":1,
            "C":1,
            "v":4,
            "H":1,
            "h":3,
            "i":4,
            "k":1,
            "_":4,
            "o":1
        },
        "3":{
            "w":1,
            "h":1,
            "_":1
        },
        "6":{
            "Q":1,
            "C":1,
            "v":1,
            "w":1,
            "z":1,
            "_":1
        },
        "7":{
            "Q":4,
            "b":2,
            "r":2,
            "C":2,
            "w":4,
            "G":2,
            "h":2,
            "H":2,
            "x":2,
            "z":4,
            "N":2
        },
        "10":{
            "Q":1,
```

```
         "C":1,
         "G":1,
         "w":1,
         "H":1,
         "x":1,
         "h":1,
         "z":1,
         "J":1,
         "_":1
      },
      "14":{
         "b":3,
         "C":3,
         "D":3,
         "G":2,
         "H":2,
         "h":3,
         "J":3,
         "N":2,
         "O":3,
         "Q":3,
         "q":1,
         "r":1,
         "v":1,
         "w":3,
         "x":3,
         "z":3,
         "_":3
      }
   },
   "readonlyMap":{
      "CustomEvent":1,
      "B":1,
      "D":1,
      "MutationObserver":1,
      "G":1,
      "N":1,
      "O":1,
```

```
      "Q":1,
      "ob":1,
      "Pc":1,
      "b":1,
      "d":1,
      "RegExp":1,
      "g":1,
      "h":1,
      "i":1,
      "NolTracker":1,
      "k":1,
      "m":1,
      "Date":1,
      "n":1,
      "tb":1,
      "o":1,
      "Array":1,
      "q":1,
      "Promise":1,
      "r":1,
      "pb":1,
      "u":1,
      "Object":1
   },
   "typestableMap":{
      "C":1,
      "G":1,
      "h":1,
      "i":1,
      "J":1,
      "pa":1,
      "Q":1,
      "HTMLImageElement":1,
      "u":1,
      "v":1,
      "w":1,
      "x":1,
      "Object":1,
```

```
      "z":1
   },
   "randomMap":{
      "Promise":1,
      "Q":1,
      "s":1,
      "H":1,
      "NolTracker":1,
      "i":1,
      "Object":1,
      "n":1,
      "o":1,
      "_":1
   }
}
```

schwab.com:

```
{
   "isSingle":false,
   "isEmpty":false,
   "webSiteType":1,
   "creatorsNumber":{
      "1":1114,
      "2":19,
      "4":1,
      "7":1
   },
   "readOnly":58,
   "typeStable":1,
   "randomChange":8,
   "numberofCreOcc":{
      "1":{
         "tt":3,
         "A":2,
         "St":1,
         "rt":8,
         "B":4,
```

```
        "Ce":1,
        "Visitor":293,
        "Ae":20,
        "jn":1,
        "AppMeasurement":583,
        "br":1,
        "HTMLImageElement":2,
        "Iterator":1,
        "ke":8,
        "Je":3,
        "AppMeasurement_Module_Media":8,
        "yt":1,
        "Boolean":7,
        "Fe":8,
        "st":1,
        "c":3,
        "d":8,
        "buildPixel":14,
        "RegExp":8,
        "Lr":3,
        "Nt":1,
        "mt":9,
        "Ye":5,
        "Date":14,
        "n":2,
        "o":5,
        "Ue":6,
        "p":3,
        "Array":5,
        "ar":1,
        "r":19,
        "t":42,
        "v":2,
        "w":6,
        "xr":1,
        "vt":1
    },
    "2":{
```

```
         "Tt":1,
         "rt":8,
         "d":8,
         "Visitor":1,
         "Ot":3,
         "hr":1,
         "AppMeasurement_Module_Media":5,
         "It":3,
         "yt":1,
         "AppMeasurement":1,
         "AppMeasurement_Module_ActivityMap":5,
         "Tr":1
      },
      "4":{
         "un":1,
         "Un":1,
         "tn":1,
         "rn":1
      },
      "7":{
         "p":1,
         "st":1,
         "a":1,
         "Qt":1,
         "s":1,
         "t":1,
         "l":1
      }
   },
   "readonlyMap":{
      "tt":1,
      "A":1,
      "St":1,
      "rt":1,
      "B":1,
      "Ce":1,
      "Qt":1,
      "Ae":1,
```

```
      "jn":1,
      "Visitor":1,
      "Ot":1,
      "hr":1,
      "It":1,
      "br":1,
      "Iterator":1,
      "un":1,
      "ke":1,
      "Je":1,
      "yt":1,
      "Tr":1,
      "Fe":1,
      "Tt":1,
      "st":1,
      "a":1,
      "c":1,
      "d":1,
      "RegExp":1,
      "Nt":1,
      "Lr":1,
      "mt":1,
      "l":1,
      "Ye":1,
      "n":1,
      "Date":1,
      "o":1,
      "Array":1,
      "p":1,
      "Ue":1,
      "ar":1,
      "s":1,
      "t":1,
      "v":1,
      "w":1,
      "Un":1,
      "xr":1,
      "tn":1,
```

```
      "rn":1,
      "vt":1
   },
   "typestableMap":{
      "HTMLImageElement":1
   },
   "randomMap":{
      "r":1,
      "Visitor":1,
      "t":1,
      "buildPixel":1,
      "AppMeasurement_Module_Media":1,
      "Boolean":1,
      "AppMeasurement":1,
      "AppMeasurement_Module_ActivityMap":1
   }
}
```

timeout.com:

```
{
   "isSingle":false,
   "isEmpty":false,
   "webSiteType":1,
   "creatorsNumber":{
      "1":521,
      "2":1,
      "3":6,
      "4":2
   },
   "readOnly":9,
   "typeStable":1,
   "randomChange":9,
   "numberofCreOcc":{
      "1":{
         "a":3,
         "b":24,
         "Visitor":203,
```

```
         "e":181,
         "RegExp":5,
         "MutationObserver":1,
         "H":1,
         "NolTracker":22,
         "Date":1,
         "o":68,
         "Promise":1,
         "R":10,
         "v":1
      },
      "2":{
         "b":1,
         "v":1
      },
      "3":{
         "b":6,
         "v":6,
         "g":6
      },
      "4":{
         "a":2,
         "r":2,
         "i":2,
         "o":2
      }
   },
   "readonlyMap":{
      "Promise":1,
      "r":1,
      "RegExp":1,
      "MutationObserver":1,
      "H":1,
      "i":1,
      "Date":1,
      "o":1
   },
   "typestableMap":{
```

```
      "o":1
   },
   "randomMap":{
      "a":1,
      "R":1,
      "b":1,
      "Visitor":1,
      "e":1,
      "v":1,
      "g":1,
      "NolTracker":1,
      "o":1
   }
}
```