Undestanding JavaScript Semantics

Peter Lozov, Dmitri Kosarev, Dmitri Boulytchev

Abstract

In this document we discuss the pecularities of JavaScript semantics and give some guidelines which would help to navigate through and understand the relavant literature.

1 Introduction

JavaScript is a general-purpose language which serves as a *de-facto* standard for web-based applications development. Historically, JavaScript was developed and released in the middle of 1990s as a mean to add dynamicity to web-pages. Language-wise, JavaScript is an imperative object-oriented dynamically typed language with first-class functions and objects. Currently JavaScript is supported by all mainstream web-browsers via different implementations: V8 [10] for Chrome-based browsers, Chakra — for Microsoft Edge, SpiderMonkey [9] — for Mozilla Firefox and some others, and, finally, JavaScriptCore [11] for Safari.

Nowdays JavaScript is a full-fledged general-purpose programming language which possesses a number of somewhat unique properties. While for a regular web programming it is often sufficient to stick with a number of tutorials [12] developing *tools* require a much greater level of internalization of JavaScript semantics. This document provides some introduction to JavaScript pecular features, references to relevant documentation and some guidelines for its reading and understanding.

The principal source of information on JavaScript is the specification [1] which is maintained by Ecma International — an European standartization organization in the domain of informatics and telecommunications. The specification spans over 800 pages and is intended to be self-sufficient. For this purpose the first sections introduce the terminological and typographical conventions used throuhout the rest of the specification as well as algorithmic notation for specifying the semantics of some built-in operations and procedures. However being written in a natural language the specification is verbose and hardly admits a formal treatment. For this purpose a number of attempts to re-formulate the semantics of the language in more conventional terms were taken. In [2, 3] a small-step operational semantics is given for a reasonably chosen subset of the language and some safety properties for JavaScript programs are proven. The resulting formal semantics (as reported by the authors)

occupies 70 pages of ASCII code. The papers also investigate the differences in implementations; for example, the following program

```
var f =
  function () {
    if (true) {function g () {return 1}}
    else      {function g () {return 2}};

    function g () {return 3};

    return g ();

    function g () {return 4}
  };
```

is reported to behave differently (as per moment of writing) under different implementations due to their different one- vs. multi-pass architectures. The function definitions in ECMAScript are *hoisted* (order-preserving moved to the top of the scope they are declared in), thus the last definition of "g" shadows all others, which is not always respected by the implementations.

Another attempt is taken in [4, 5] where a certified implementation of JavaScript in Coq is presented. The development is accompanied with an extracted reference interpreter and a number of tests. The results reported are reproducible after a reasonal amount of efforts [6].

It's worth noticing that both mentioned results deal with minor versions of ECMAScript (ECMAScript-4 and ECMAScript-5 respectively), thus they do not support some more recent features of the language such as, for example, classes. Moreover, even within the addressed standard not all features are supported which makes their direct usage as a reference implementation problematic.

In the subsequent sections we address some properties of JavaScript semantics which we consider the most interesting; we label the description with the exact references to the ECMAScript specification given in a [designated] form.

2 Objects

Objects [6.1.7] constitute the most important kind of values JavaScript programs operate with. In a nutshell, objects are finite collections of *properties*. The specification makes a distinction between *data* properties and *accessor* properties. Each property possesses a *key* which in fact can be a *symbol* (indentifier), a string or an integer, and a *value* which is a regular JavaScript value. Thus, objects are finite maps. In order to make it possible to specify objects explicitly JavaScript introduces a dedicated notation — JSON (JavaScript Object Notation). For example, in the following snippet

```
let obj = {name: "object"; id : 1};
```

an explicit object with two properties named “name” and “id” with values ”object” and 1 respectively is defined; a reference to this object is placed in the variable obj.

The essential property of JavaScript objects is their dynamic nature. Properties can be added, removed, and modified at a runtime. In the context of the previous example one can make

```
obj.name = 3;
obj.kind = "new property example";
delete obj.id;
```

changing the value (and a type) of the property “name”, adding a new property “kind” with corresponding value and deleting the property “id”. Additionally, each property is equipped with a set of *attributes* [6.1.7.1] which controls what operations are permitted for this property. These attributes can be set at a runtime, too.

The specification distinguishes *ordinary* [10.1] objects from *exotic* [10.4] ones. The difference between the two lays in different behaviour of some built-in operations. For example, arrays ans strings are exotic since they accept integers as names for their properties.

All objects form a hierarchy via a *prototype* inheritace (see Section 3). The basis of this hierarchy is Object object [20.1] which defines the properties common for all objects. In the context of previous example one may perform a call

```
obj.toString ()
```

although ”obj” does not possess “toString” as its own property. This property is “borrowed” from Object. The call returns string `"[object Object]"` in a full accordance with the specification [20.1.3.6].

The language is equipped with a control construct [14.7.5] which, in particular, allows for iterating over all properties of objects. For example the construct

```
for (let p in obj) {
  ...
}
```

goes over all *own* properties of an object “obj”, binding identifier “p” to the name of a property on each iteration. The property's value then can be accessed via construct “obj [p]”.

3 Functions, Constructors, and Prototypes

JavaScript is equipped with full-fledged first-class functions with closures. Beside that, however, functions possess a number of pecularities.

First, functions are regular objects. For example, given a definition

```
let f = function (x) {return x;};
```

one may make a call

```
f.toString ()
```

which returns a string `"function (x) {return x;}"`. Since (as is it rather natural to anticipate) " toString " is a function as well it can be called on itself

```
f.toString.toString ()
```

which returns `"function toString() { [native code] }"`. Function objects [20.2] occupy a dedicated place in objects' hierarchy and share a number of common properties. For example

```
f.length
```

gives a number of arguments which function "f" accepts.

Then, functions play role of *constructors* [6.1.7.2]. For example, the following definition

```
function Person (name, age, address) {
  this.name    = name;
  this.age     = age;
  this.address = address;
}
```

allows for creating a similarly structured objects *en masse*:

```
let Alice      = new Person ("Alice", 12, "Wonderland");
let MadHatter  = new Person ("Mad Hatter", undefined, "Tea Party");
```

It is worth mentioning that there is nothing special about function "Person" in comparison with other functions. Its role as a constructor is primarily conventionalized by a usage together with the construct "**new**". Internally, JavaScript sometimes can tell constructor functions apart from non-constructor ones (for example, for built-in functions or functions explicitly defined as class constructors), but for regular user-defined functions it can not do this. Therefore, there is nothing wrong with a plain call

```
let x = Person ("Global Object", 0, "Right Here");
```

However, in this case corresponding properties will be added or set for the *global object* [19] by the discipline of binding for "**this**" (see Section 5).

Similarly, a function

```
function foo (x, y) {return x+y;}
```

can be coupled with "**new**":

```
let p = new foo (2, 3);
```

The result associated with "p", however, will be an empty Object rather then number 5.

Finally, function objects possess a property called "prototype" which is used for implementing inheritance. Each object holds an implicit reference to

the "prototype" property of a constructor this object was created by. In this regard constructor functions play role of "types": Object is not a "type" in JavaScript, but a function:

```
Object.toString ()
```

gives back the string `"function Object() { [native code] }"`, and similarly for Function, String, Number, Date and all other "standard types".

As prototypes are regular objects their set of properties can be manupulated with. We show this by the following example: given the definitions for Alice and MadHatter shown above we can do

```
Alice.mood = "cheerful";
```

Now Alice and MadHatter have different sets of properties: Alice has its own property "mood" while MadHatter has not (which can be easily tested). On the other hand, we could add the "mood" property to the *prototype* of Person:

```
Person.prototype.mood = "indifferent";
```

Now all instances of Person acquire given property with given value; in particular

```
MadHatter.mood
```

is now evaluated to `"indifferent"`. However,

```
Alice.mood
```

is still `"cheerful"` since it refers to Alice's *own* property which overrides that in the prototype. Deleting this property

```
delete Alice.mood
```

reverts Alice.mood to `"indifferent"` since now the resolution procedure ends up in the prototype.

Prototype properties can be modified only for prototype as whole. For example,

```
MadHatter.mood = "gloomy"
```

does not change Alice.mood since it introduces MadHatter's own property.

In overall, the resolution process for reading a property of an objects first tries to find its own property and if the property is not found it recurses in its prototype; writing a property creates an own property of an object or changes its value if such an own property already exists.

Modification of prototypes makes it possible to introduce global changes in program behaviour. For example

```
Object.prototype.toString = function () {JSON.stringify (this);};
```

changes the default way in which objects are rendered as strings.

4 Methods, Inheritance, and Late Binding

Object-oriented constructs which are built in many conventional object-oriented languages can be simulated in JavaScript using mechanisms described in the previous sections.

Methods are functional properties of objects:

```
let obj = {
  name     : "Alice",
  surname  : "Liddell",
  fullname : function () {return this.name + " " + this.surname;}
};
```

Methods can also be defined in constructors:

```
function Person (name, surname) {
  this.name     = name;
  this.surname  = surname;
  this.fullname =
    function () {
      return this.name + " " + this.surname;
    };
}
```

Using prototypes some default methods can be added for all instances created by a constructor:

```
Person.prototype.abbrevname =
  function () {return this.name[0] + ". " + this.surname;};
```

This "default" implementation can be overridden in a concrete instance:

```
let Alice = new Person ("Alice", "Liddell");

Alice.abbrevname = function () {return "The Alice";};
```

These examples work as expected by the very dynamic nature of property resolution in JavaScript.

Finally, inheritance can be simulated by manipulating with prototypes. Let us reiterate the last example:

```
function Person (name, surname) {
  this.name     = name;
  this.surname  = surname;
  this.fullname =
    function () {
      return this.name + " " + this.surname;
    };
  this.abbrevname =
    function () {
      return this.name[0] + ". " + this.surname;
```

```
    };
}
```

Now we derive a new kind of objects which borrow some methods from "Person" and override some others:

```
function Traveller (name, surname, destination) {
  this.name        = name;
  this.surname     = surname;
  this.destination = destination;
  this.abbrevname  =
    function () {
      return this.name[0] + ". " + this.surname + " to " + destination;
    }
}

Traveller.prototype = new Person ("", "");
```

By setting the prototype of Traveller to the "default" Person instance we make the JavaScript to search for non-overriden methods.

We can also make an observation that in JavaScript, as in any language with first-class functions, objects can be simulated in many other ways even not involving built-in object-oriented constructs like prototypes or "**this**" [13].

5 The Magic of "this"

As we've seen in the previous sections the behaviour of constructors and methods essentially depends on the binding rules for "**this**" keyword:

1. in an object method, "**this**" refers to the object;
2. alone, "**this**" refers to the global object;
3. in a non-method function, "**this**" refers to the global object.

The exact description is given in [9.4.4].

References

[1] ECMAScript ® 2025 Language Specification. Draft ECMA-262, March 7, 2024 // `https://tc39.es/ecma262/`

[2] Sergio Maffeis, John C. Mitchell, Ankur Taly. An Operational Semantics for JavaScript. // Asian Symposium on Programming Languages and Systems, 2008, pp. 307-325, `https://seclab.stanford.edu/websec/jsPapers/aplas08-camera-ready.pdf`

[3] Sergio Maffeis, John C. Mitchell, Ankur Taly. An Operational Semantics for JavaScript, revised and extended version // http://jssec.net/semantics/sjs.pdf

[4] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, Gareth Smith. A Trusted Mechanised JavaScript Specification // POPL'14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 2014, pp. 87–100.

[5] A Coq Specification of ECMAScript 5 (JavaScript) with Verified Reference Interpreter // jscert.org

[6] A Coq Specification of ECMAScript 5 (JavaScript) with Verified Reference Interpreter, reflections on building, installation, and run // https://github.com/Lozov-Petr/jscert

[7] Ecma International. Industry Association for Standardizing Information and Communication Systems // https://ecma-international.org

[8] ChakraCore project website // https://github.com/chakra-core/ChakraCore

[9] SpiderMonkey project website // https://spidermonkey.dev

[10] V8 project website // https://chromium.googlesource.com/v8/v8

[11] JavaScriptCore project website // https://docs.webkit.org/Deep%20Dive/JSC/JavaScriptCore.html

[12] JavaScript Tutorial // https://www.w3schools.com/js/

[13] Rajesh Naroth. JavaScript Inheritance without ES6 classes // https://rajeshnaroth.medium.com/javascript-inheritance-without-es6-classes-6ff546c0d58b