Programming in Javascript
A guide

by Ramon Wenger
at SCG
http://scg.unibe.ch
University of Bern

Abstract

The purpose of this document is to provide anyone with the goal of writing his own Javascript code with some hopefully helpful general tips, caveats and indications of common error sources in relation to the language that is Javascript.
This guide claims in no way to be exhaustive or to contain everything there is to know about Javascript. It's built on the experience of the author, collected while programming his own application.
This document should also not be mistaken as a tutorial to Javascript, as there are a lot of basic programming concepts that will not be discussed here and the explanation of which would go beyond the scope of this report.

Table of contents

Introduction

After being nothing more than a gimmick that didn't have its place in most websites except as a tool for the notorious and frowned upon pop up advertisements for a long time, Javascript has made its ascension from almost nothing to being used in the most successful online appearances of our time. Javascript forms a trinity with **HTML/XML** and **CSS**, and while **HTML/XML** (**H**yper**T**ext **M**arkup **L**anguage/e**X**tensible **M**arkup **L**anguage) is responsible for laying the structure of a website, or more specifically, for creating the **DOM** (**D**ocument **O**bject **M**odel), and **CSS** (**C**ascading **S**tyle **S**heets) defines, what a website looks like, **Javascript** interacts with both of them and extends what can be done with them in many ways.
Javascript is a programming language with many similarities to other languages. People who already have had contact with other languages might recognize familiar concepts, but could also be surprised by some little differences which could prevent a desired outcome.

Variables and Types

Even though every variable has its own type, Javascript handles its types generally rather loosely.
Variables are declared via the keyword *var*. No type is declared.
Also, when declaring functions, specifically their arguments, no type safety is enforced.
There is an operator named *typeof*, with which the type of a variable can be read.
Usage:

```
typeof variable
```

The possible *return* values are

```
number
string
boolean
object
function
undefined
```

Of special note should be the type *undefined*. A variable that has not been defined yet reads as *undefined*, which is not the same as *null* (which has the type *object*), like it would be in many other languages.
undefined is also the value returned when a function ends without *return* value, for example

```
function someFunc(){
      return;
}
```

Also noted should be a special value for a number variable, and that is *NaN*, which stands for 'Not a Number', and is the result of an operation which should return a number, but can't be converted to one. Many other languages convert such a value to *0* or *null*, so beware of that.

Something all these values have in common though is, when used as a boolean, they produce *false*, for example in an *if*-statement. The list of values that produce a *false* are:

```
false
null
undefined
'' (empty string)
0
NaN
```

Functions have their own type in Javascript, and a function can be assigned to a variable.
For example

```
var one = function(){
      return 1;
}
```

is the same as

```
function one(){
      return 1;
}
```

where

```
typeof one
```

produces "*function*", and

```
typeof one()
```

produces "*number*" in this example, and in general the type of the return value of the function.

Variables and Scope

Variables can exist in one of two scopes.
One of them is when they're being defined in a function, with or without the keyword *var*, and therefore have a local scope. They are forgotten as soon as the function returns, in the general case.
The other one is the global scope, and variables created in this scope are always assigned to their parent. This may be evident when looking at variables declared inside objects, for example

```
myObject = {
      aString : "hello"
}
```

where *aString* can be later referred to as *myObject.aString*.
What's more interesting is, that a variable that's declared outside of an object also has a parent, and if not specified otherwise, this is always the *window* variable, the parent element of the DOM.
This means, that calling a function not belonging to an object, e.g. *one()* from before, is the same as calling *window.one()*
In combination with another quirk of Javascript, the usage of *this*, this fact can lead to some frustrating errors.
this is always declared at the time when calling a function, and refers to the object calling said function.
Example:

```
function two() {
      return this;
}
```

If we call *two()*, we really call *window.two()*, which in turn returns the object *window*.
On the other hand, if two is assigned to another object, like

```
myObject.myFunc = two
```

and *myObject.myFunc()* is called, *myObject* is returned.

This is important, amongst other things, when assigning functions to certain events in an HTML file.
Take

```
<button id="b1"></button>
<button id="b2" onclick="someFunc();"></button>
<script type="text/javascript">
b1 = document.getElementById("b1");
b1.onclick = someFunc;
someFunc(){
      return this;
}
</script>
```

Example adapted from: [1]

Clicking on the two buttons here would return button *b1* in case of the first button, but *window* in the case of the second button, as the functions called are *b1.onclick*() and *window.someFunc()*, respectively

There are two native Javascript functions that can provide a way around this.
One of them is *apply*, the other one is *call*, and they're both methods of the *function* prototype.
Usage:

```
someFunc.call(obj, argument1, argument2, argument3, ...)
```

and

```
someFunc.apply(obj, argumentArray)
```

They basically do both the same thing, just where one takes a comma separated list of arguments, the other accepts an array. They both call the function they're invoked on (*someFunc* here), set *this* to the first argument (*obj* here), and use the rest of the arguments as arguments for the function. So

```
someFunc = function(a,b){
      alert(a+", ",b);
      return this;
}
someFunc.call(obj, "hello", "world");
```

would pop up an alert box with "**hello, world**" and then return the object *obj.*

Arrays

One important thing that's different in Javascript is the handling of the *foreach* loop

```
for (arr in array)
```

While in a lot of other languages this would loop through the array, with *arr* holding each element of the array iteratively, in Javascript the variable *arr* simply holds the name of the current key of the array.
So

```
for(a in array){
	a.someMethod();
}
```

wouldn't work, one would have to call

```
for(a in array){
	array[a].someMethod();
}
```

Objects

Javascript is a classless language, but there is a sort of simulation of classes.
An object can be generated with a call of a constructor function with the *new* keyword.
Example:

```
function Point(xVal, yVal){
      this.x = xVal;
      this.y = yVal;
}
```

When we then do this

```
p1 = new Point(0,0);
```

p1 is assigned an object of the type *Point*, with a value for *p1.x* and one for *p1.y*

If we were to omit the keyword *new* and just call

```
p1 = Point(5,10);
```

p1 would be assigned the value *undefined*, as this is the default *return* value for functions without an actual *return* value, and *window* would get two new properties, *window.x* and *window.y.*

To be noted here is also the fact that objects in Javascript are always passed by reference, and never copied.
If we do the following

```
p2 = p1;
p2.x = 5;
```

then *p1.x* will also be changed, as *p1* and *p2* are just different references to the same object.
Unfortunately there is no native function in Javascript that copies an object as it is. It is left up to the programmer to write a function that fulfills this task. A general solution for any object is very difficult, if not outright impossible, to define.
The main difficulty with a general solution lies with the inability to decide which attributes of an object should be copied by value and therefore with the ability to differ from the original object's attributes, and which of them should be just copied by reference, so that changes to the original object can be reflected on the copy.

Closure and functions

It is of utmost importance to understand one thing when developing pure Javascript, and that's closure. A good definition is

> *A closure is basically a function/method that has the following two properties:*
>
> - *You can pass it around like an object (to be called later)*
> - *It remembers the values of all the variables that were in scope when the function was created. It is then able to access those variables when it is called even though they may no longer be in scope.*

Source: [2]

That means, when calling a function inside another function, the inner function still has access to the variables defined outside of it, in the outer function. This can be a blessing, but also a curse if one doesn't pay attention to what this means exactly.
A closure can, for example, be used to create an object with properties that can't be accessed by anyone but the functions of that same object.
Example:

```
f1 = function(){
      var myVar = 5;
      var inc = function(arg){
            myVar += arg;
            return myVar;
      }

      return inc;
}();
```

Source: [3]

Notice the parentheses at the end of the declaration. With them, the function declared just before is invoked right after its declaration. With this kind of call used here, there will be no way for the function just invoked to be accessed later as such, just for what's generated by it, i.e. its return value. *f1* is set to the function *inc* at that time. When we invoked for example

```
f1(5);
```

this will return *10*, even though *myVar* already theoretically shouldn't exist anymore in the global scope, because the original function concluded. But it still does exist, because the inner function references it. That's called a closure

```
f1.myVar;
```

will return *undefined* though, because in this global scope it simply is not defined.

Closures can prove treacherous too, for if we do something like this

```
a = [];
for(i=0;i<10;i++){
      a[i] = function(){ return i };
}
```

we will notice, when calling any element of the array

```
a[5]();
```

it will return *10*, no matter what index of the array we use. This is because the functions still reference the value of *i* at the end of the loop, which is still in scope for this function.
If we want every function to have the value of *i* at the time of the declaration of the function, we have to use a closure in such a way that *i* isn't referenced anymore, by creating another inner function and calling it right away

```
b = [];
for(i=0;i<10;i++){
      b[i] = function() {return (function(){ return i;})()};
}
```

This way the outer function resolves at the time of declaration, and the inner function will reference the value of i it had at that point in time.

Closures are most likely the one thing that's the most exciting for someone taking a deeper look into Javascript for the first time, and not having been introduced to the concept by another language using them before. They can be very confusing, but at the same time, when understood well, also a very rewarding tool.

Conclusion

Javascript is a very interesting and fun language, especially when understood in its own little peculiarities. For a deeper look into the mannerisms of the language, the reader is referenced to the O'Reilly book **JavaScript, The Good Parts [3]**, and the web in general, as there is an abundance of good resources to be found, especially since Javascript is one of the main languages used in the very foundation of the Internet.

The reader is advised though to also take a good look at different frameworks, like e.g. jQuery **[4]** or Prototype **[5**], as they provide new functionality to the language and also do away with some of the quirks of the language. Armed with a basic knowledge of Javascript and the advanced tools these frameworks provide, many of the possible challenges of the language should offer little resistance.

References / Sources / Further reading

[1] http://devlicio.us/blogs/sergio_pereira/archive/2009/02/09/javascript-5-ways-to-call-a-function.aspx

[2] http://www.skorks.com/2010/05/closures-a-simple-explanation-using-ruby/

[3] Crockford Douglas, 2008, JavaScript: The Good Parts, O'Reilly, 172 p.

[4] http://jquery.com/

[5] http://www.prototypejs.org/