As you know, JavaScript is the number one programming language in the world, the language of the web, of mobile hybrid apps (like PhoneGap or Appcelerator), of the server side (like NodeJS or Wakanda) and has many other implementations. It's also the starting point for many new developers to the world of programming, as it can be used to display a simple alert in the web browser but also to control a robot (using nodebot, or nodruino). The developers who master JavaScript and write organized and performant code have become the most sought after in the job market.

In this article, I'll share a set of JavaScript tips, tricks and best practices that should be known by all JavaScript developers regardless of their browser/engine or the SSJS (Server Side JavaScript) interpreter.

Note that the code snippets in this article have been tested in the latest Google Chrome version 30, which uses the V8 JavaScript Engine (V8 3.20.17.15).

1 – Don't forget `var` keyword when assigning a variable's value for the first time.

Assignment to an undeclared variable automatically results in a global variable being created. Avoid global variables.

2 – use `===` instead of `==`

The `==` (or `!=`) operator performs an automatic type conversion if needed. The `===` (or `!==`) operator will not perform any conversion. It compares the value and the type, which could be considered faster than `==`.

```
[10] === 10    // is false
[10]  == 10    // is true
'10' == 10     // is true
'10' === 10    // is false
 []   == 0     // is true
 [] ===  0     // is false
 '' == false   // is true but true == "a" is false
 '' ===   false // is false
```

3 – `undefined`, `null`, `0`, `false`, `NaN`, `''` (empty string) are all falsy.

4 – Use Semicolons for line termination

The use of semi-colons for line termination is a good practice. You won't be warned if you forget it, because in most cases it will be inserted by the JavaScript parser. For more details about why you should use semi-colons, take a look to this artice: http://davidwalsh.name/javascript-semicolons.

5 – Create an object constructor

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

var Saad = new Person("Saad", "Mousliki");
```

6 – Be careful when using `typeof`, `instanceof` and `constructor`.

- *typeof* : a JavaScript unary operator used to return a string that represents the primitive type of a variable, don't forget that `typeof null` will return "object", and for the majority of object types (Array, Date, and others) will return also "object".
- *constructor* : is a property of the internal prototype property, which could be overridden by code.
- *instanceof* : is another JavaScript operator that check in all the prototypes chain the constructor it returns true if it's found and false if not.

```
var arr = ["a", "b", "c"];
typeof arr;   // return "object"
arr  instanceof Array // true
arr.constructor();  //[]
```

7 – Create a Self-calling Function

This is often called a Self-Invoked Anonymous Function or Immediately Invoked Function Expression (IIFE). It is a function that executes automatically when you create it, and has the following form:

```
(function(){
    // some private code that will be executed automatically
})();
(function(a,b){
    var result = a+b;
    return result;
})(10,20)
```

8 – Get a random item from an array

```
var items = [12, 548 , 'a' , 2 , 5478 , 'foo' , 8852, , 'Doe' , 2145 , 119];

var  randomItem = items[Math.floor(Math.random() * items.length)];
```

9 – Get a random number in a specific range

This code snippet can be useful when trying to generate fake data for testing purposes, such as a salary between min and max.

```
var x = Math.floor(Math.random() * (max - min + 1)) + min;
```

10 – Generate an array of numbers with numbers from 0 to max

```
var numbersArray = [] , max = 100;

for( var i=1; numbersArray.push(i++) < max;);  // numbers = [1,2,3 ... 100]
```

11 – Generate a random set of alphanumeric characters

```
function generateRandomAlphaNum(len) {
    var rdmString = "";
    for( ; rdmString.length < len; rdmString  += Math.random().toString(36).substr(2));
    return  rdmString.substr(0, len);

}
```

12 – Shuffle an array of numbers

```
var numbers = [5, 458 , 120 , -215 , 228 , 400 , 122205, -85411];
numbers = numbers.sort(function(){ return Math.random() - 0.5});
/* the array numbers will be equal for example to [120, 5, 228, -215, 400, 458, -85411, 122205]  */
```

A better option could be to implement a random sort order by code (e.g. : Fisher-Yates shuffle), than using the native sort JavaScript function. For more details take a look to this discussion.

13 – A string trim function

The classic trim function of Java, C#, PHP and many other language that remove whitespace from a string doesn't exist in JavaScript, so we could add it to the `String` object.

```
String.prototype.trim = function(){return this.replace(/^s+|s+$/g, "");};
```

A native implementation of the trim() function is available in the recent JavaScript engines.

14 – Append an array to another array

```
var array1 = [12 , "foo" , {name "Joe"} , -2458];

var array2 = ["Doe" , 555 , 100];
Array.prototype.push.apply(array1, array2);
/* array1 will be equal to  [12 , "foo" , {name "Joe"} , -2458 , "Doe" , 555 , 100] */
```

15 – Transform the `arguments` object into an array

```
var argArray = Array.prototype.slice.call(arguments);
```

16 – Verify that a given argument is a number

```
function isNumber(n){
    return !isNaN(parseFloat(n)) && isFinite(n);
}
```

17 – Verify that a given argument is an array

```
function isArray(obj){
    return Object.prototype.toString.call(obj) === '[object Array]' ;
}
```

Note that if the toString() method is overridden, you will not get the expected result using this trick.

Or use…

```
Array.isArray(obj); // its a new Array method
```

You could also use `instanceof` if you are not working with multiple frames. However, if you have many contexts, you will get a wrong result.

```
var myFrame = document.createElement('iframe');
document.body.appendChild(myFrame);
```

```
var myArray = window.frames[window.frames.length-1].Array;
var arr = new myArray(a,b,10); // [a,b,10]

// instanceof will not work correctly, myArray loses his constructor
// constructor is not shared between frames
arr instanceof Array; // false
```

18 – Get the max or the min in an array of numbers

```
var  numbers = [5, 458 , 120 , -215 , 228 , 400 , 122205, -85411];
var maxInNumbers = Math.max.apply(Math, numbers);
var minInNumbers = Math.min.apply(Math, numbers);
```

19 – Empty an array

```
var myArray = [12 , 222 , 1000 ];
myArray.length = 0; // myArray will be equal to [].
```

20 – Don’t use delete to remove an item from array

Use `splice` instead of using `delete` to delete an item from an array. Using `delete` replaces the item with `undefined` instead of the removing it from the array.

Instead of…

```
var items = [12, 548 ,'a' , 2 , 5478 , 'foo' , 8852, , 'Doe' ,2154 , 119 ];
items.length; // return 11
delete items[3]; // return true
items.length; // return 11
/* items will be equal to [12, 548, "a", undefined × 1, 5478, "foo", 8852, undefined × 1, "Doe", 2154,       119]   */
```

Use…

```
var items = [12, 548 ,'a' , 2 , 5478 , 'foo' , 8852, , 'Doe' ,2154 , 119 ];
items.length; // return 11
items.splice(3,1) ;
items.length; // return 10
/* items will be equal to [12, 548, "a", 5478, "foo", 8852, undefined × 1, "Doe", 2154,       119]   */
```

The delete method should be used to delete an object property.

21 – Truncate an array using length

Like the previous example of emptying an array, we truncate it using the `length` property.

```
var myArray = [12 , 222 , 1000 , 124 , 98 , 10 ];
myArray.length = 4; // myArray will be equal to [12 , 222 , 1000 , 124].
```

As a bonus, if you set the array length to a higher value, the length will be changed and new items will be added with `undefined` as a value. The array length is not a read only property.

```
myArray.length = 10; // the new array length is 10
myArray[myArray.length - 1] ; // undefined
```

22 – Use logical AND/ OR for conditions

```
var foo = 10;
foo == 10 && doSomething(); // is the same thing as if (foo == 10) doSomething();
foo == 5 || doSomething(); // is the same thing as if (foo != 5) doSomething();
```

The logical OR could also be used to set a default value for function argument.

```
function doSomething(arg1){
    arg1 = arg1 || 10; // arg1 will have 10 as a default value if it’s not already set
}
```

23 – Use the map() function method to loop through an array’s items

```
var squares = [1,2,3,4].map(function (val) {
    return val * val;
});
// squares will be equal to [1, 4, 9, 16]
```

24 – Rounding number to N decimal place

```
var num =2.443242342;
num = num.toFixed(4);  // num will be equal to 2.4432
```

NOTE : the `toFixed()` function returns a string and not a number.

25 – Floating point problems

```
0.1 + 0.2 === 0.3 // is false
9007199254740992 + 1 // is equal to 9007199254740992
9007199254740992 + 2 // is equal to 9007199254740994
```

Why does this happen? 0.1 +0.2 is equal to 0.30000000000000004. What you need to know is that all JavaScript numbers are floating points represented internally in 64 bit binary according to the IEEE 754 standard. For more explanation, take a look to this blog post.

You can use `toFixed()` and `toPrecision()` to resolve this problem.

26 – Check the properties of an object when using a for-in loop

This code snippet could be useful in order to avoid iterating through the properties from the object's prototype.

```
for (var name in object) {
    if (object.hasOwnProperty(name)) {
        // do something with name
    }
}
```

27 – Comma operator

```
var a = 0;
var b = ( a++, 99 );
console.log(a);  // a will be equal to 1
console.log(b);  // b is equal to 99
```

28 – Cache variables that need calculation or querying

In the case of a jQuery selector, we could cache the DOM element.

```
var navright = document.querySelector('#right');
var navleft = document.querySelector('#left');
var navup = document.querySelector('#up');
var navdown = document.querySelector('#down');
```

29 – Verify the argument before passing it to `isFinite()`

```
isFinite(0/0) ; // false
isFinite("foo"); // false
isFinite("10"); // true
isFinite(10);   // true
isFinite(undefined);  // false
isFinite();   // false
isFinite(null);  // true  !!!
```

30 – Avoid negative indexes in arrays

```
var numbersArray = [1,2,3,4,5];
var from = numbersArray.indexOf("foo") ;  // from is equal to -1
numbersArray.splice(from,2);    // will return [5]
```

Make sure that the arguments passed to `splice` are not negative.

31 – Serialization and deserialization (working with JSON)

```
var person = {name :'Saad', age : 26, department : {ID : 15, name : "R&D"} };
var stringFromPerson = JSON.stringify(person);
/* stringFromPerson is equal to "{"name":"Saad","age":26,"department":{"ID":15,"name":"R&D"}}"   */
var personFromString = JSON.parse(stringFromPerson);
/* personFromString is equal to person object  */
```

32 – Avoid the use of `eval()` or the `Function` constructor

Use of `eval` or the `Function` constructor are expensive operations as each time they are called script engine must convert source code to executable code.

```
var func1 = new Function(functionCode);
var func2 = eval(functionCode);
```

33 – Avoid using `with()` (The good part)

Using `with()` inserts a variable at the global scope. Thus, if another variable has the same name it could cause confusion and overwrite the

value.

34 – Avoid using for-in loop for arrays

Instead of using…

```
var sum = 0;
for (var i in arrayNumbers) {
    sum += arrayNumbers[i];
}
```

…it's better to use…

```
var sum = 0;
for (var i = 0, len = arrayNumbers.length; i < len; i++) {
    sum += arrayNumbers[i];
}
```

As a bonus, the instantiation of `i` and `len` is executed once because it's in the first statement of the for loop. Thsi is faster than using…

```
for (var i = 0; i < arrayNumbers.length; i++)
```

Why? The length of the array `arrayNumbers` is recalculated every time the loop iterates.

NOTE : the issue of recalculating the length in each iteration was fixed in the latest JavaScript engines.

35 – Pass functions, not strings, to `setTimeout()` and `setInterval()`

If you pass a string into `setTimeout()` or `setInterval()`, the string will be evaluated the same way as with `eval`, which is slow. Instead of using…

```
setInterval('doSomethingPeriodically()', 1000);
setTimeout('doSomethingAfterFiveSeconds()', 5000);
```

…use…

```
setInterval(doSomethingPeriodically, 1000);
setTimeout(doSomethingAfterFiveSeconds, 5000);
```

36 – Use a switch/case statement instead of a series of if/else

Using switch/case is faster when there are more than 2 cases, and it is more elegant (better organized code). Avoid using it when you have more than 10 cases.

37 – Use switch/case statement with numeric ranges

Using a switch/case statement with numeric ranges is possible with this trick.

```
function getCategory(age) {
    var category = "";
    switch (true) {
        case isNaN(age):
            category = "not an age";
            break;
        case (age >= 50):
            category = "Old";
            break;
        case (age <= 20):
            category = "Baby";
            break;
        default:
            category = "Young";
            break;
    };
    return category;
}
getCategory(5);  // will return "Baby"
```

38 – Create an object whose prototype is a given object

It's possible to write a function that creates an object whose prototype is the given argument like this…

```
function clone(object) {
    function OneShotConstructor(){};
    OneShotConstructor.prototype= object;
    return new OneShotConstructor();
```

```
}
clone(Array).prototype ;  // []
```

39 – An HTML escaper function

```
function escapeHTML(text) {
    var replacements= {"<": "&lt;", ">": "&gt;","&": "&amp;", """: "&quot;"};
    return text.replace(/[<>&"]/g, function(character) {
        return replacements[character];
    });
}
```

40 – Avoid using try-catch-finally inside a loop

The try-catch-finally construct creates a new variable in the current scope at runtime each time the catch clause is executed where the caught exception object is assigned to a variable.

Instead of using…

```
var object = ['foo', 'bar'], i;
for (i = 0, len = object.length; i <len; i++) {
    try {
        // do something that throws an exception
    }
    catch (e) {
        // handle exception
    }
}
```

…use…

```
var object = ['foo', 'bar'], i;
try {
    for (i = 0, len = object.length; i <len; i++) {
        // do something that throws an exception
    }
}
catch (e) {
    // handle exception
}
```

41 – Set timeouts to `XMLHttpRequests`

You could abort the connection if an XHR takes a long time (for example, due to a network issue), by using `setTimeout()` with the XHR call.

```
var xhr = new XMLHttpRequest ();
xhr.onreadystatechange = function () {
    if (this.readyState == 4) {
        clearTimeout(timeout);
        // do something with response data
    }
}
var timeout = setTimeout( function () {
    xhr.abort(); // call error callback
}, 60*1000 /* timeout after a minute */ );
xhr.open('GET', url, true);

xhr.send();
```

As a bonus, you should generally avoid synchronous XHR calls completely.

42 – Deal with WebSocket timeout

Generally when a WebSocket connection is established, a server could time out your connection after 30 seconds of inactivity. The firewall could also time out the connection after a period of inactivity.

To deal with the timeout issue you could send an empty message to the server periodically. To do this, add these two functions to your code: one to keep alive the connection and the other one to cancel the keep alive. Using this trick, you'll control the timeout.

Add a `timerID`…

```
var timerID = 0;
function keepAlive() {
    var timeout = 15000;
    if (webSocket.readyState == webSocket.OPEN) {
```

```
        webSocket.send('');
    }
    timerId = setTimeout(keepAlive, timeout);
}
function cancelKeepAlive() {
    if (timerId) {
        cancelTimeout(timerId);
    }
}
```

The `keepAlive()` function should be added at the end of the `onOpen()` method of the webSocket connection and the `cancelKeepAlive()` at the end of the `onClose()` method.

43 – Keep in mind that primitive operations can be faster than function calls. UseVanillaJS.

For example, instead of using…

```
var min = Math.min(a,b);
A.push(v);
```

…use…

```
var min = a < b ? a : b;
A[A.length] = v;
```

44 – Don’t forget to use a code beautifier when coding. Use JSLint and minification (JSMin, for example) before going live.

45 – JavaScript is awesome: Best Resources To Learn JavaScript

- Code Academy JavaScript tracks: http://www.codecademy.com/tracks/javascript
- Eloquent JavaScript by Marjin Haverbeke: http://eloquentjavascript.net/
- Advanced JavaScript by John Resig: http://ejohn.org/apps/learn/

Conclusion

I know that there are many other tips, tricks and best practices, so if you have any ones to add or if you have any feedback or corrections to the ones that I have shared, please adda comment.

References

In this article I have used my own code snippets. Some of the snippets are inspired from other articles and forums:

- JavaScript Performance Best Practices (CC)
- Google Code JavaScript tips
- StackOverFlow tips and tricks
- TimeOut for XHR