

Dos and don't in Javascript

Javascript good parts and what to avoid

Azat Satklichov

azats@seznam.cz,

<http://sahet.net/htm/java.html>,

<https://github.com/azatsatklichov/java-in-deep>

JS History, Ecosystem



JS Engines:

[V8](#) from [Google](#)
[SpiderMonkey](#) by [Mozilla](#)
[JavaScriptCore](#) from [Apple](#)
[Chakra](#) by [Microsoft](#)
[Tamarin](#) from [Adobe Flash](#).
[Rhino & Nashorn](#) by [Oracle Java](#)

2020	Chrome	Edge/IE	Firefox	Safari	Opera
October	80.4 %	5.2 %	7.1 %	3.7 %	2.1 %

Year	Chrome	IE	Firefox	Safari	Opera
2015	63.3 %	6.5 %	21.6 %	4.9 %	2.5 %

2009	6.5 %	39.4 %	47.9 %	3.3 %	2.1 %
2008		52.4 %	42.6 %	2.5 %	1.9 %
2007		58.5 %	35.9 %	1.5 %	1.9 %
				Netscape	
2006		62.4 %	27.8 %	0.4 %	1.4 %
2005		73.8 %	22.4 %	0.5 %	1.2 %
				Mozilla	
2004		80.4 %	12.6 %	2.2 %	1.6 %
2003		87.2 %	5.7 %	2.7 %	1.7 %
2002		84.5 %	3.5 %	7.3 %	

Languages for Web: HTML/CSS, **JavaScript**, Java applets, Flash/Flex

1993 Mosaic - first browser

1994 - Netscape Navigator

1995 – **Mocha** -> Live Script(JS) -> JavaScript (Brendan Eich, 10 days dev.)

1996 Browser war started: Microsoft **JScript** for **IE** browser

1997 – **ECMAScript** (spec.): JavaScript, JScript, ActionScript

2005 – **AJAX** born (E4X)

2008 – ES4 (Yahoo, Google, Microsoft, ..) -> ES6

2009 – CommonJS: Browserless JS, way to **Node.js**.

2015-2018 versions of JavaScript (ES6 - ECMAScript 2015)

Libs/Frameworks: jQuery, MooTools, D3.js, ExtJS, Angular, Vue, React,

..

UnitTesting: Unit.js, Jasmine, Jest, Mocha, Protractor, Cypress, ..

JS Alternatives: in writing code / transpilers: DART, **Typescript**, AtScript, Elm, ClojureScript, Kaffeine, CoffeeScript, BottomLine, Opal, Roy, ...

Alongside client and server software, it is now even possible to write native mobile apps using JavaScript. Also one language for FE&BE&DB

What brings Typescript (compile time check,)

- Typed Superset of JS, transpiled to JS (no type information). ES5 | ES6 ,
- Static type, Composing types, Generic, REST args,
- OOP features: Interfaces, Classes, Enums, ..
- More ...

Do's and Don'ts: <https://www.typescriptlang.org/docs/handbook/declaration-files/do-s-and-don-ts.html>

Don't ever use the types Number, String, Boolean, Symbol, or Object

/ WRONG */*

function reverse(s: String): String;

/ OK */*

function reverse(s: string): string;

Return Types of Callbacks: Don't use the return type any for callbacks whose value will be ignored:

/ WRONG */*

function fn(x: () => any) { x(); }

/ OK */ (even VOID is problematic: not type, discards value, ..)*

function fn(x: () => void) { x(); }

It still does not solve the problems caused by JS, ...

JavaScript is a sloppy language, and does a lot error-forgiveness. JSLint/ESLint/TSLint or TS or libraries like (lodash, decimal.js, ..) helps you to program better JS code and to avoid most of the slop.

Best Practices

- Avoid Global Variables: Global variables and functions can be overwritten by other scripts
- Beware of Automatic Type Conversions: "3"+"2"=32, "3"-1=2 Confusing Addition & Concatenation
- Use Parameter Defaults: otherwise undefined if not provided
- Avoid using eval() : run text as code, security issue, like bat or shell files
- Misunderstanding Floats: var y = 0.2; var z = x + y // result in z will not be 0.3;
- Avoid creating objects via **new** on Number, String, Boolean, Objects, Array, ..
- Accessing Arrays with Named Indexes:
 - properties will produce undefined or incorrect results:
- Undefined is Not Null
- Be ready to expect weird results
- Reduce Activity in Loops
- Reduce DOM Size
- Avoid Using with
- Reduce DOM Access: Accessing the HTML DOM is slow
- ES5/6, TS, JSLint/EsLint, new libraries like lodash ,..

```
console.log("3" + "2");//32
console.log("3" - "2");//1
```

```
var x = "Hello";
var y = new String( value: "Hello");
(x === y)// false, x is a string and y is an object.
```

```
//Even worse
var x = new String( value: "Hello");
var y = new String( value: "Hello");
(x == y)// is false because you cannot compare objects.
```

```
var person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
var x = person.length; // person.length will return 0
var y = person[0]; // person[0] will return undefined
```

```
//Browser compatibility issues
var str = "HELLO WORLD";
str[0]; //IE 7 or earlier, ...
str[0] = "A";// Gives no error, but does not work
/str[0]; // returns H
```

The Problematic/Sloppy part of JavaScript - reason to avoid

- It's weakly typed, which makes the code comparatively obscure;
- Code is open, can be used for malicious purposes and compromise the client-side security;
- Code has to be tested on different browsers, won't execute/interpret in the same way on every browser;
- Drawbacks including [IEEE 754 Double Precision](#), the fact that a number of functions and properties tend to be [executed differently across browsers](#), [aggressive coercion](#), and [problematic global variables](#) (to name just a few).

JavaScript Data, Object types:

Primitive Data Types: string, number, boolean, symbols(es6), null, undefined. Primitive values are **immutable**

Types of Objects: [type object] Object, Date, Array, String, Number, Symbol, Boolean, function[function]. Mutatable key-col.

Types contain no values: null, undefined [typeof null is **object**], [typeof undefined is **undefined**]

HOISTING

JavaScript Declarations are Hoisted. Variables can be used before declared. It is JS's default behavior of moving all **declarations** regardless of where a function is placed, to the top of the scope. JavaScript only hoists declarations, not **initializations**.

```
x = 5; // Assign 5 to x
//some logic
var x;
```

```
<script>
var x = 5; // Initialize x
elem = document.getElementById("demo"); // Find an element
elem.innerHTML = "x is " + x + " and y is " + y; // Display x and y
var y = 7; // Initialize y
</script>
```

```
//eg. const and let
carName = "Volvo";
document.getElementById("demo").innerHTML = carName;
let carName; //ReferenceError: Cannot access 'carName' before initialization
```

The function Statement, function Expression - fun-statements (not fun-expression) are subject to hoisting.

```
function foo( ) {}
```

```
var foo = function foo( ) {};
```

Objects JS objects inherit members from the prototype chain so they are *never truly empty*. To test for membership without prototype chain involvement, use the `hasOwnProperty` method or limit your results

new - creates a new object that inherits from the operand's prototype member, and then calls the operand, binding the new object to this. **Avoid creating Strings, Numbers, Booleans, Objects, & Arrays as Objects!** Unexpected results and slow down execution speed. Two objects can't be even compared. There is no compile-time or runtime warning.

```
var a = new String('b');    //String object
var b = new Number(10);     //Number object
var c = new Boolean(true);  //Boolean object
var d = new Object();       //Object object
var e = new Array();        //Array object
```

Use: var a = "b"; var b = 10; var c = true; var d= {}; var e=[] instead

```
let check = new Boolean (false);
if(check) {
    console.log("WOW FOUND, even FALSE");//FOUND, even FALSE
}

let check2 = false;
if(check2) {
    console.log("WOW FOUND, even FALSE");//skipped
}
```

this - keyword refers to the object it belongs to. It has different values depending on where it is used:

- In a method, **this** refers to the **owner object**.
- Alone, **this** refers to the **global object**, [object Window].
- Also, in "use strict" **this** refers to the **global object** [object Window].
- In a function, **this** refers to the **global object**. [object Window].
- In a function, in strict mode, **this** is **undefined**.
- In an event, **this** refers to the **element** that received the event.
- Methods like `call()`, and `apply()` can refer **this** to **any object**.
- In short, with arrow functions there are no binding of **this**.



Phony Arrays - slower than real arrays

JavaScript does not have real arrays. No need to give them a dimension, and they never generate outof-bounds errors. But their performance can be considerably worse than real arrays.

The **typeof operator does not distinguish between arrays and objects.**

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
```

```
typeof fruits; //object, how can I recognize ARRAY
```

1) `isArray` function or

```
2) if (my_value && typeof my_value === 'object'
      && typeof my_value.length === 'number'
      && !(my_value.propertyIsEnumerable('length'))) { //my_value is
    definitely an array!
  }
```

Another example

```
var elms = new Array(40,100); //Creates an array with two elements (40 and 100)
```

```
var elms = new Array(40); // Creates an array with 40 undefined elements!!!!
```

```
var person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
var x = person.length; // person.length will return 3
var y = person[0];     // person[0] will return "John"
```

re-define arrays with named-indexes

```
var person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
var x = person.length; // person.length will return 0
var y = person[0];     // person[0] will return undefined
```

hasOwnProperty When using a *for in* loop, usually a good idea to use **hasOwnProperty(variable)** to make sure the property belongs to the object you want and is not instead an inherited property from the prototype chain:

```
for (myvariable in object) {
  if
  (object.hasOwnProperty(myvariable)) {
    ... //statements to be executed
  }
}
```

hasOwnProperty - can be replaced with other function

```
var name;
another_stooge.hasOwnProperty = null; // trouble
for (name in another_stooge) {
  if (another_stooge.hasOwnProperty(name)) { // boom
    document.writeln(name + ': ' + another_stooge[name]);
  }
}
```

Semicolon insertion When inserting semicolons in places where they are not welcome, returns **undefined** ;)

```
return
{
  status: true
}; //returns undefined
```

```
return {
  status: true
}; //returns { status: true }
```

null and undefined

Undefined refers to something which has not yet been assigned a value (yet).
Null refers to something which definitively has no value. In that case, just return a null. Also different browsers seem to be returning these differently.

null and undefined differences

```
typeof undefined //undefined
typeof null      // object
null == undefined // true
null === undefined // false
```

void - In JavaScript, void is an operator (but in many languages, **void is a return type**) that takes an operand and returns undefined. This is not useful, and it is very confusing. **Avoid void.**

```
console.log(void 0); //undefined
console.log(undefined); //undefined
```

```
// a naughty global variable from another script
var undefined = "oops";
```

ES5 improved this by making undefined immutable

void - introduced to handle the shadowing of undefined in **pre-ES5**, where undefined on the global scope can be **overridden** or **shadowed**. Is an operator not return type.

```
3) void function what() {
  console.log('What')
}();
//using void for IIFE will always evaluate to undefined.
what();
```

```
1) let i = void 2; // i === undefined
```

```
2) <a href="javascript:void(0)">Click Here</a>
```

```
4) let tt = (function() {})(()) === void 0; //undefined
console.log(tt); //true
```

VOID method returns UNDEFINED. //In Java VOID method does not return anything, ...

Global Variables should be avoided

- Visible in every scope, can significantly complicate the behavior of the program, harder to run subprograms
- Can be re-declared, re-assigned, ...

Three ways to define global variables

- **var foo = value;** (any place outside the function)
- **window.foo = value;** (add directly to global object)
- **foo = value;** (implied global or implicit global via variable hoisting) more open to buggy programs

JS has function SCOPE: Each function creates new scope. Variables declared in function, become **LOCAL** to the function. Undefined Variables and Functions. If a variable is not explicitly declared, then JS assumes that the variable was global

E.g.1

```
function foo() {  
    var variable1, variable2;  
    variable1 = 5; variable2 = 6;  
    return variable1 + variable2;  
}
```

E.g.2

```
var carName = "Ford";  
console.log(carName);  
var carName; //still has value Ford  
console.log(carName);  
var carName = 76; //type changed  
console.log(carName);
```

Alternative:

- To minimize use of global-var: var myApp = {}; which holds all files for your app. e.g. Angular style...
- Use let|const (scoped var) .
- Use tslint/eslint: helps to catch variable-hoisting bugs, to find if you use variable out of scope
- Or closure for data hiding

Typeof, instanceof - returns a string that identifies the type

```
console.log( typeof 0 ); // "number"
console.log( typeof new Number() ); // "object"
console.log( typeof Infinity ); // "number"
console.log( typeof NaN ); // "number"
console.log( typeof 'Hi' ); // "string"
console.log( typeof new String("Hi") ); // "Object"
console.log( typeof true ); // "boolean"
console.log( typeof new Boolean(true) ); // "object"
console.log( typeof null ); // "object", not null
console.log( typeof undefined ); // "undefined"
//console.log( typeof new Symbol() ); // "symbol" - only in ES6
console.log( typeof [] ); // "object" - arrays are objects.
console.log( typeof [1,2,3] ); // "object", not "array"
console.log( typeof new Array(1,2,3) ); // "object"
console.log( typeof new Date() ); // "object"
console.log( typeof /^$/ ); // "object"
console.log( typeof new RegExp('^$') ); // "object"
console.log( typeof {a: "abc"} ); // "object"
console.log( typeof typeof {} ); // "string"
console.log( typeof function () {} ); // "function"
```

All objects are *truthy* and null is *falsy*

Arrays are objects in JavaScript so typeof array will return 'object'

You cannot use typeof to determine if a JS object is an array (or a date).

```
var myArr = [1,2,3,4];
console.log(myArr.constructor.toString().indexOf("Array") > -1);
console.log(myArr.constructor === Array ); // 2-way
isArray // 3 way, ...
```

Falsy Values - These values are all **falsy**, but **they are not interchangeable**.

All other values are **truthy** including all objects & the string 'false'.

```
if (my_value && typeof my_value === 'object') {  
  //then my value is definitely an object or an array  
  //and truthy (first statement)  
}
```

```
let check = new Boolean (false);  
if(check) {  
  console.log("WOW FOUND, even FALSE");//FOUND, even FALSE  
}
```

All objects are *truthy* and null is *falsy*

```
let check2 = false;  
if(check2) {  
  console.log("WOW FOUND, even FALSE");//skipped  
}
```

Value	Type
0	Number
NaN (not a number)	Number
' ' (empty string)	String
false	Boolean
null	Object
undefined	Undefined

Instanceof checks if an object is an instance of a class or constructor

```
//array is complex type, even though [] has typeof Object  
console.log([] instanceof Array); // true  
console.log(typeof []); //object
```

```
//instance of only works for complex data type  
const a = "I'm a string primitive";  
const b = new String("I'm a String Object");  
console.log(a instanceof String); //returns false  
console.log(b instanceof String); //returns true
```

E.g. shows clarity on the differences between **typeof** and **instanceof**

```
const isString = (str) => {  
  return typeof str === 'string' || str instanceof String  
}
```

1. **== (equal values)** 2. **=== (both type and value, but still not complete)** 3. **Object.is()**

Comparing two JavaScript objects will **always** return **false**.

//result is same with number/Number, boolean/Boolean, array/Array

```
var x = "JavaScript";
```

```
var y = new String("JavaScript");
```

```
x == y //true, equal values
```

```
x === y //false, have different types (string and object)
```

== operator always converts (to matching types) before comparison.

=== operator forces comparison of values and type:

```
5 == '5' //true
```

```
5 === "5" //false
```

Even worse: Objects cannot be compared:

```
var x = new String("JavaScript");
```

```
var y = new String("JavaScript");
```

```
// (x == y) is false because x and y are different objects
```

```
// (x === y) is false because x and y are different objects
```

```
// just if two obj same, NaN can be used during comparison
```

```
Object.is(x, x); //true
```

```
Object.is(x, y); //false
```

```
0 == ""; // true
```

```
1 == "1"; // true
```

```
1 == true; // true
```

```
0 === ""; // false
```

```
1 === "1"; // false
```

```
1 === true; // false
```

Manual comparison needed to compare Objects:

Read properties and compare them manually. Eg. `object1.name === object2.name`

You also can use `Object.keys()` [shallow(for primitives) or deep versions] just to fasten the process.

Other ways: ES6 way `Object.entries(k1).toString() === Object.entries(k2).toString()`; No for nested obj+keys order

2) `JSON.stringify` (key order matters) 3) **LODASH**

In JAVA .. Just `equals()` checks value equality, `==` if both references to same OBJ in memory

parseInt – no warning or informative message when converting also issue is with dates/times/

Eg1: `parseInt("32")` and `parseInt("32 meters")` produce the same result 32,

Eg2: `parseInt("08")` and `parseInt("09")` based on Base 8, but es5 (integer) ;) **Provide Radix.**

Floating point (64bit), no warning|info message, behaves differently in different browser converts a string into an integer.

0.1 + 0.2 is not equal to 0.3 `//0.30000000000000004` `==> var z = (x * 10 + y * 10) / 10; //0.3`

Numbers in JS no types of numbers, like integers, short.. numbers without a period or exp. are accurate up to 15 digits

`var x = 9999999999999999; // x will be 9999999999999999`

`var y = 9999999999999999; // y will be 10000000000000000`

```
console.log(0.1 + 0.2); //0.30000000000000004
console.log(0.1 + 0.2 === 0.3); //false // ???
x = 1.0000000000000001
console.log(x === 1); //true // ???
```

Bitwise – JavaScript has the same set of bitwise operators (&, |, ^, ~, >>, ..) as Java: In Java they work with integers. JS

has no integers. Bit operator work on 32bit, result converted back to JS, ...

In JavaScript, they are very far from the hardware and very slow. JavaScript is rarely used for doing bit manipulation. As a result, in JavaScript programs, it is more likely that & is a mistyped && operator.

NaN – NaN (Not-a-Number) is not equal to any value (including itself, **NaN !== NaN**) and is essentially an **illegal number value**, but `typeof(NaN)===number //true`. Use `isNaN(number)` to check for NaNs

```
var isNumber = function isNumber(value) {
  return typeof value === 'number' && isFinite(value); //not NaN & Infinity
```



THANK YOU

References

https://www.w3schools.com/js/js_intro.asp

<https://github.com/dwyl/JavaScript-the-Good-Parts-notes>

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

<https://medium.com/javascript-non-grata/the-top-10-things-wrong-with-javascript-58f440d6b3d8>