

Using objects in JavaScript. Accessing DOM in JavaScript

Vyacheslav Koldovskyy
Last update: 29/03/2016

Agenda

- Program flow control
- Collections
- Custom objects
- Constructors
- Context and "*this*"
- Operator "*new*"
- Browser Object Model (BOM) and Document Object Model (DOM)
- Events
- Memory and Sandbox
- Closures

Conditions: if-else

Syntax:

```
if (condition)
    statement1
[else
    statement2]
```

Example:

```
var age = Number(prompt('Please enter your age', 0));
if (age < 16) {
    alert('You are underage!')
} else {
    alert('You are adult!')
}
```

<https://jsfiddle.net/koldovsky/dkc3gn79/>

Conditional (Ternary) Operator ?:

Syntax:

```
condition ? expr1 : expr2
```

Example:

```
var age = Number(prompt('Please enter your age', 0));  
var msg = (age < 16) ? 'underage' : 'adult';  
alert('You are ' + msg + '!');
```

<https://jsfiddle.net/koldovsky/pujawr71/>

Loops: for

Syntax:

```
for ([initialization]; [condition]; [final-expression])  
    statement
```

Example:

```
for (var i = 0; i <= 10; i++) {  
    console.log(i);  
}
```

<https://jsfiddle.net/koldovsky/boc1w3rm/>

One processing of loop's body is called **iteration**.

Loops: while and do-while

```
var i = 0;
while (i <= 10) {
    console.log(i);
    i++;
}
```

```
var i = 0;
do {
    console.log(i);
    i++;
} while (i <= 10)
```

<https://jsfiddle.net/koldovsky/1v7yobmo/>

<https://jsfiddle.net/koldovsky/2gaad0mg/>

- The main difference between these loops is the moment of condition calculation.
- *While* calculates condition, and if the result is true, *while* does iteration.
- *Do-while* initially does iteration and after that calculates a condition.

Loops: keywords break and continue

There are two keywords for loops control :

- **break** – aborts loop and moves control to next statement after the loop;
- **continue** – aborts current iteration and immediately starts next iteration.

Try not to use this keywords. A good loop have one entering point, one condition and one exit.

Switch

Switch statement allows to select one of many blocks of code to be executed. If all options don't fit, default statements will be processed

```
var mark = Number(prompt('Enter mark between 1 and 5', 1));
var text;
switch (mark) {
  case 1: text = 'very bad';
    break;
  case 2: text = 'bad';
    break;
  case 3: text = 'satisfactorily';
    break;
  case 4: text = 'good';
    break;
  case 5: text = 'excellent';
    break;
  default: text = 'incorrect';
}
alert('Your mark is ' + text);
```

<https://jsfiddle.net/koldovsky/dr5cy28j/>

Collections

Collection is a set of variables grouped under common name.

Usually elements of collections are grouped according to some logical or physical characteristic.

Collections help to avoid situations when we have to declare multiple variables with similar names::

var a1, a2, a3, a4...

There are two types of collections that are typical for JS: arrays and dictionaries (hash tables).

Array: processing

Usage of arrays:

```
var array = [] // declaration of empty array
```

```
var array = [5, 8, 16] // declaration of predefined array
```

```
array[0] = 4; // writing value with index 0
```

```
tmp = array[2]; // reading value by index (in tmp - 16)
```

```
array.length // getting length of array
```

Array: features

- Arrays in JavaScript differ from arrays in classical languages.
- Arrays in JS are instances of Object.
- So Array in JS can be easily resized, can contain data of different types and have string as an index.
- Length of array is contained in length property, its value is equal to **index of last element increased by one.**

Array: useful methods

Some useful methods of array:

array.push(value) – add element to the end of an array

array.pop() – extract element from end of an array

array.unshift(value) – insert element before first

array.shift() – extract first element

array.join() – concatenate all elements into a string

string.split() – split a string into an array of substrings

array.sort() – built-in method to sort array

Iterating an Array

```
var arr = ['H', 'e', 'l', 'l', 'o'];  
for (var i = 0; i < arr.length; i++) {  
    console.log(arr[i]);  
}
```

<https://jsfiddle.net/koldovsky/0d697kaL/>

```
var arr = ['H', 'e', 'l', 'l', 'o'];  
arr.forEach(function(el, i) {  
    console.log(el);  
});
```

<https://jsfiddle.net/koldovsky/whmv60cn/>

Dictionary

- Dictionaries allow us to have set of data in form of "key-value" pairs
- We can create hash and initialize it at the same time. For this we should write values separated by a comma like in array. But for all values we have to set key:

```
var name = {  
    key: value,  
    key: value  
};
```

This format of describing of JS object with the only exception – it requires double quotes, has its own name: JavaScript Object Notation or short **JSON**.

Using Dictionary

Usage of dictionaries tables is very similar to arrays:

```
dict['good'] = 4; // writing value in element with key "good"  
tmp = dict['excellent']; // reading value by key "excellent"
```

The difference is in usage of **for-in** statement:

```
for (key in dict) {  
    console.log(dict[key]);  
}
```

Array vs Dictionary

Use **Array** for collections with digital indexes.

Use **Hash** if you want use string keys.

Don't look for property *length* in **Hash**.

Don't look for *forEach* and other **Array** methods in **Hash**.

Always explicitly declare **Array** otherwise you get a **Hash**.

Don't use *for* with hash, use *for-in* instead.

At finally : use collection – be cool :)

Object creation

You know that we can create a simple object in JavaScript. We use JSON for this.

```
var cat = { [1]  
  name: 'Snizhok',  
  color: 'white'  
};
```

Object or Dictionary

But this way it looks like hash table creation. What is the difference between hash table and object, then?

```
var hash = {  
  key: value,  
  key: value  
};
```

[1]

?

```
var object = {  
  key: value,  
  key: value  
};
```

Object or Dictionary

Typically we use hash table if we want to represent some collection, and we use an object to describe some system or entity.

```
var cats = {  
  first: murzyk,  
  second: barsyk  
};
```

[1]

!

```
var cat = {  
  name: barsik,  
  color: white  
};
```

Difference in use

There are some differences in using of hash tables and objects as a result. For example:

```
cats['first']; // good way [1]
```

To access elements of hash table we use indexer [] with key inside. But it's incorrect for objects! For objects Operator "." should be used :

```
cat['name']; // incorrect!  
cat.name; // good way [2]
```

Constructors

Sometimes we need to create more than one single object. It is not a good idea to use the literal way for this. It will be better create a *scenario* for objects reproducing.

Constructor is a function that implements this scenario in JavaScript.

Constructor consists of declaration attributes and methods that should be added into each new object with presented structure.

Constructors: example

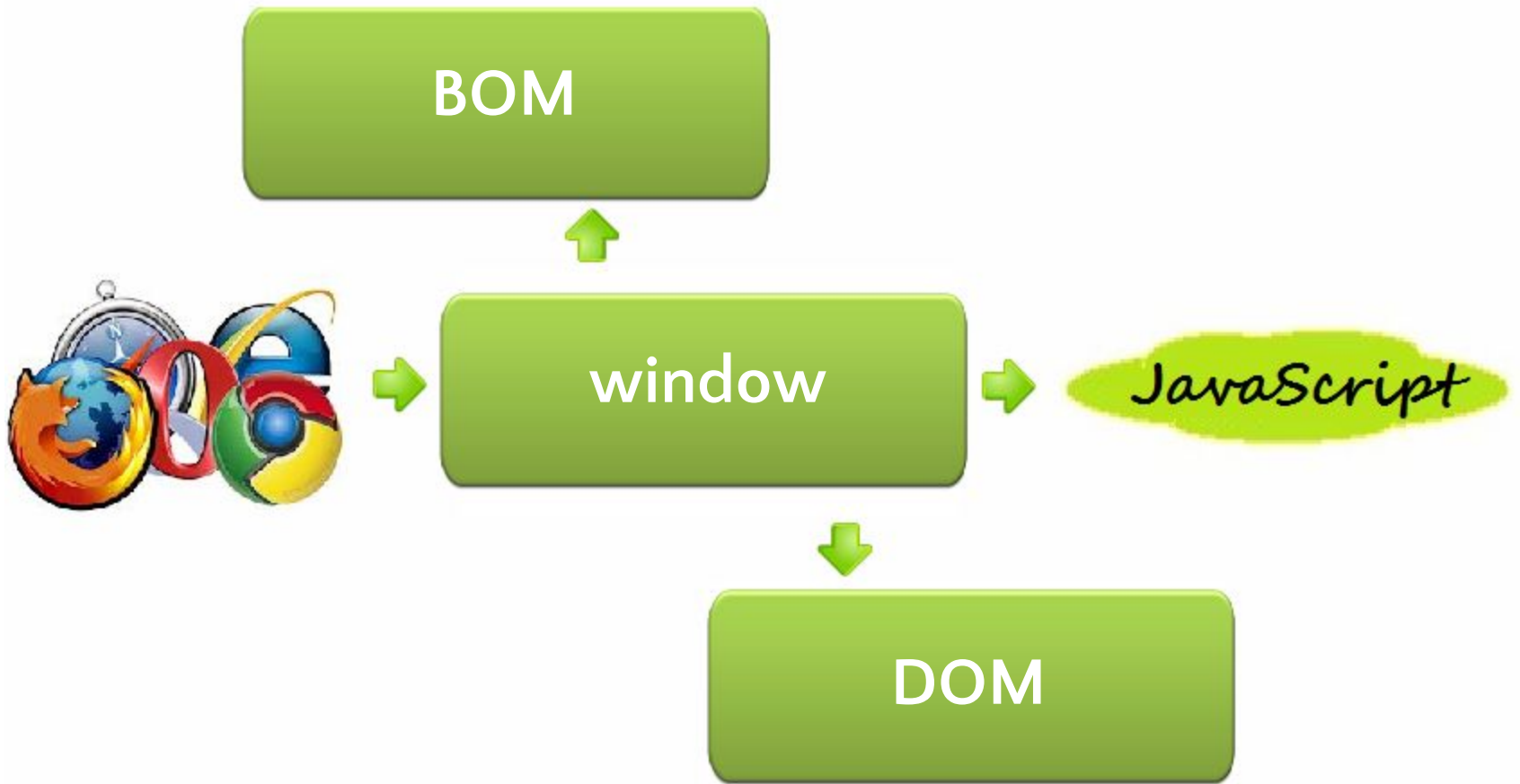
```
function Cat (name) {  
  this.name = name;  
  this.run = function () {  
    console.log(this.name + ' run!');  
  };  
  return this;  
}
```

[1]

```
var murzyk = new Cat('Murzyk');
```

[2]

BOM and DOM



Description

How JavaScript communicates with the world?

In outline this mechanism works by next scenario: user does something and this action is an event for browser. JavaScript observes pages in the browser. And if event has occurred, script will be activated.



Event handling

But JavaScript doesn't observe events by default. You should specify to your code what events are interesting for it.

There are 3 basic ways to subscribe to an event:

- inline in HTML
- using of *onevent* attribute
- using special methods

First and second ways are deprecated for present days. Let's take a look at event handling in more details.

Inline handling

Imagine that we have some HTML-element, for example `<button>` and we want to do some action when user clicks the button.

First way: inline adding of JavaScript into HTML. If we use this technique, we should update HTML-page and set some JS code in *onclick* attribute of HTML-element.

```
<button onclick = "action();" > Demo </button>
```



[1]

Never use this way, because it influences HTML and JavaScript simultaneously. So let's look at the next option!

[2]

Using of *onevent* attribute

The next way doesn't touch HTML. For adding event handler you need to find an object that is a JavaScript model of HTML-element.

For example, your button has id *btn*:

```
<button id = "btn"> Demo </button>
```

[1]

Then desired object will be created automatically. Next you can use an *onclick* property:

```
btn.onclick = action;
```

Where *action* is some function defined as ***function*** *action* () { ... }

Proper ways

Previous way makes sense, but has some limitations. For example you can not use more than one handler for one event, because you set a function on *onevent* attribute directly.

Next method helps solve this and some other problems:

```
btn.addEventListener('click', action, false);
```

But this method doesn't work in IE. For IE you should use:

```
btn.attachEvent('onclick', action);
```

Proper ways

Also, you can unsubscribe from any event. In W3C:

```
btn.removeEventListener('click', action);
```

Interesting note

Why we refer to W3C if JavaScript syntax is specified by ECMA? Because ECMA specifies only cross-platform part of language and does not describes any API. The browser API is determined by W3C standards. It applies to events, DOM, storages, etc.

Bubbling and Capturing

The third parameter of **addEventListener** is a phase of event processing. There are 2 phases:

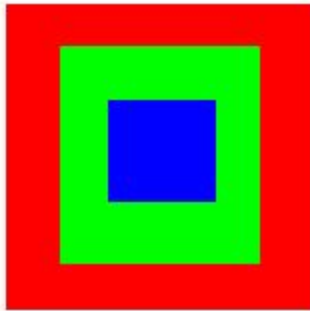
- *bubbling* (if parameter is *false*) [1]
- *capturing* (if parameter is *true*).

W3C browsers supports **both** phases whereas in **IE** only bubbling is supported.

For example:

There are three nested elements like <red>, <green> and <blue> (<div> or something else). When event has occurred inside the element <blue> its processing starts from top of DOM - window and moves to the target element. After being processed in target element event will go back.

Bubbling and Capturing

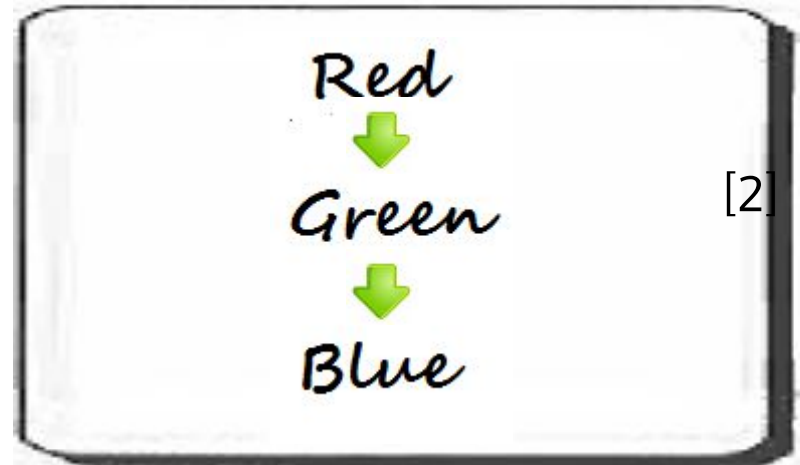


```
<red>  
  <green>  
    <blue />  
  </green>  
</red>
```

[1]

Bubbling

Capturing



Event object

For every event in the browser instance of **Event** object will be created.

You can take it if you need. In W3C browsers this object will be passed as a *first parameter* of event handler:

```
btn.addEventListener('click', action, false);
```

[1]



Where *action* was defined as:

```
function action (e) { ... }
```


Control of Default behavior

Sometimes a default scenario of event processing includes some additional behavior: bubbling and capturing or displaying context menu.

If you don't need a default behavior, you can cancel it. Use object *event* and next methods for this purpose:

`e.preventDefault();`



for aborting default browser behavior.

[2]

`e.stopPropagation();`



for discarding *bubbling* and *capturing*.

[1]

Sample

<pre>1 <div id="container"> 2 <button id="btn">Click Me!</button> 3 </div> 4</pre>	HTML ⚙️	1
<pre>1 function divClicked(e) { 2 console.log('Div clicked!'); 3 } 4 function btnClicked(e) { 5 console.log('Button clicked!'); 6 } 7 var capturing = true; 8 container.addEventListener('click', divClicked, capturing); 9 btn.addEventListener('click', btnClicked, capturing);</pre>	JAVASCRIPT ⚙️	<div data-bbox="1489 434 1663 486">Click Me!</div>

<https://jsfiddle.net/koldovsky/4rb1czbx/2/>

Practice Task

Advanced

Context and "this"

Context

Let's imagine two identical objects.
They are created by **Cat** constructor:

```
var murzyk = new Cat("Murzyk"),  
    barsyk = new Cat("Barsyk");
```

[1]

Context

If we call method `run()` for both cats, we'll take correct results:

```
murzyk.run();
```



In console:
Murzyk run!

[1]

```
barzyk.run();
```



In console:
Barsyk run!

How does the interpreter distinguish whose name should be printed?

Context

It works because we use the next form of access to attribute name: *this.name*.

this contains inside a reference to object on whose behalf was called method *run*.

Such a reference is called a **context**.

The context determined automatically after the method calling and can't be changed by code.

Loss of context

Be careful! There are situations when you can lose a context. For example:

```
setTimeout(murzyk.run, delay);
```



In console: [1]
undefined run!

murzyk.run is a reference to method. And **only reference** was saved in `setTimeout`. When the method was called by saved reference, object *window* will be used as a context and *this.name* (equal to *window.name*) was not found.

Memory and Sandbox

Basic info

Free space in browser sandbox is allocated for each variable in JavaScript.

Sandbox is a special part of memory that will be managed by browser: JavaScript takes simplified and secure access to "memory", browser translates JS commands and does all low-level work.

As a result memory, PC and user data has protection from downloaded JavaScript malware.

Scope

The scope is a special JavaScript object which was created by browser in the sandbox and used for storing variables.

Each function in JavaScript has its own personal scope. Scope is formed when a function is called and destroyed after the function finishes.

This behavior helps to manage local variables mechanism.

Object **window** is a top-level scope for all default and global variables.

Scope

```
var a = 10;  
test();  
function test () {  
  a = 30;  
  var b = 40;  
}  
var b = 20;  
console.log(a, b);
```

[1]

```
window_scope = {  
  test: function,  
  a: 10,  
  b: 20  
};
```

[2]

[41]

```
test_scope = {  
  b: 40  
};
```

[3]

Value-types and Reference-types

Unfortunately some objects are too large for scope. For example string or function. There is simple division into *value-types* and *reference-types* for this reason.

Value-types are stored in scope completely and for reference-types only reference to their location is put in scope. They themselves are located in place called "memory heap".

String and all Objects are reference-types. Other data types are stored in scope.

Memory cleaning

The basic idea of memory cleaning: when function is finished, scope should be destroyed and as a result all local variables should be destroyed.

This will work for value-types.

As for reference-types: deleting the scope destroys only reference. The object in heap itself will be destroyed only when it becomes unreachable.

Unreachable links

An object is considered unreachable if it is not referenced from the client area of code.

Garbage collector is responsible for the cleanup of unreachable objects.

It's a special utility that will launch automatically if there isn't enough space in the sandbox.

If an object has at least one reference it is still reachable and will survive after memory cleaning.

Unreachable links

```
function action () {  
    var a = new Point(10, 20),  
        b = new Point(15, 50);  
}
```

[1]

```
action_scope = {  
    a: reference,  
    b: reference  
};
```

[2]

... somewhere in heap ...

{x: 10, y: 20}

{x: 15, y: 50}

[3]

Closures

Closure


If scope is an object and it is not deleted it is still reachable, isn't it?

Absolutely! This mechanism is called **closure**.

If you save at least one reference to scope, all its content will survive after function finishing.

Example

```
function getPi () { [1]
    var value = 3.14;
    return function () { [2]
        return value;
    };
}
```



```
var pi = getPi(); [3]
...
L = 2*pi()*R;
```

Contacts

Europe Headquarters

52 V. Velykoho Str.
Lviv 79053, Ukraine

Tel: +380-32-240-9090

Fax: +380-32-240-9080

E-mail: info@softserveinc.com

Website: www.softserveinc.com

US Headquarters

12800 University Drive, Suite 250
Fort Myers, FL 33907, USA

Tel: 239-690-3111

Fax: 239-690-3116

Thank You!