SoftServe

Empowering your Business
through Software Development

Module 5:
JavaScript in Browser

D. Petin

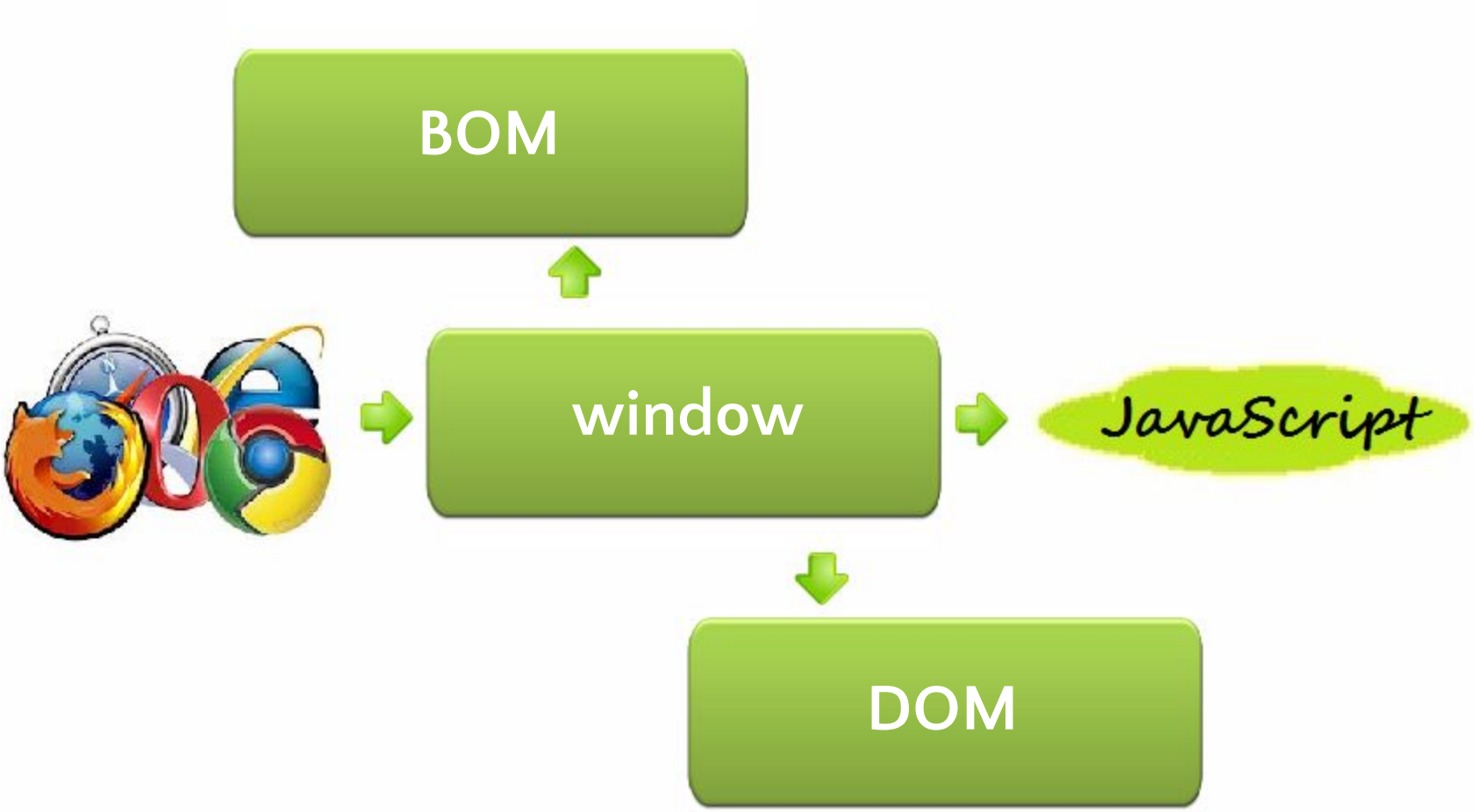
07/2014

Agenda

- JS in Browser [1]
- Events [2]
- Memory [3]
- Closure [4]

JavaScript in Browser

JavaScript in Browser



Events

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.

[1]

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:[1]

- 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.[2]

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 *onevent* 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;
```

[2]

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);
```

[1]

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

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

[2]

Proper ways

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

```
btn.removeEventListener("click", action);
```
[1]

In IE:

```
btn.detachEvent("onclick", action);
```
[1]

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'*)
- *capturing* (if parameter is *'true'*).

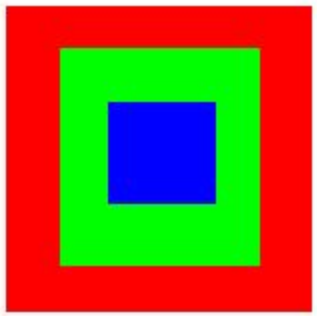
[1]

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

For example:

There are three nested elements like <red>, <green> and <blue> (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.

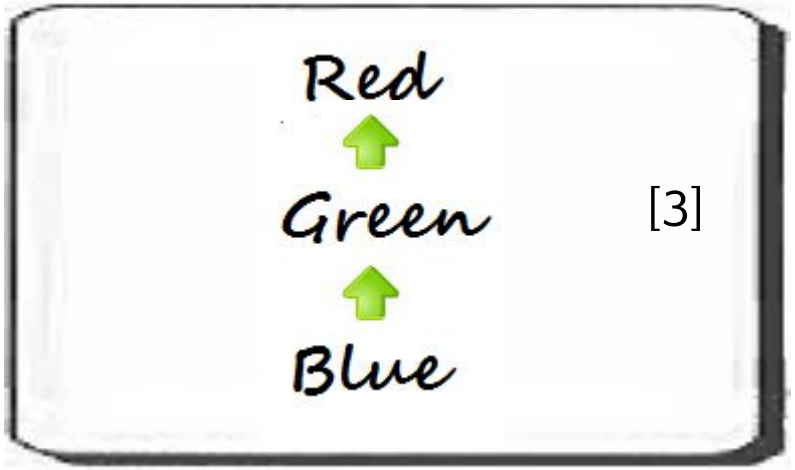
Bubbling and Capturing



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

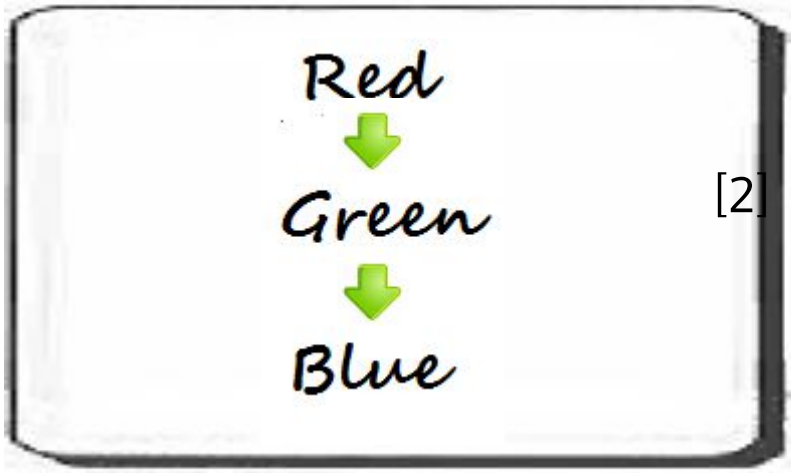
[1]

Bubbling



[3]

Capturing



[2]

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) { . . . }
```

Event object

Event object is supported in IE, too, but it's located in object *window* and its name is *event*:

```
var e = window.event;
```

[1]

You have a possibility to use a cross-browser solution.:

```
function action (e) {
    e = e || window.event;
    …
}
```

[2]

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]

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.

window object 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

FYI: 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 () {
    var value = 3.14;
    return function () {
            return value;
        };
}
```

[1]

[2]

```
var pi = getPi();
…
L = 2*pi()*R;
```

[3]


THANK
YOU