Wypieść swój JS

Spis treści

> Ten tutorial jest już niestety dość przestarzały i opisuje starszą wersję standardu JS (ECMAScript 5) a także niektóre praktyki, których obecnie nawet ja nie uznaję już za najlepsze. Dlatego lepiej potraktuj go wyłącznie jako ciekawostkę historyczną… i poczekaj aż napiszę o JS coś równie sensownego.

Inteligentny parser

JS to język, który pozwala chyba na największą dowolność w składni ze wszystkich języków programowania. Ba, jego parser jest tak sprawny, że sam sobie wstawia średniki w miejscach, w których być powinny.

```
return
{};
```

Ten krótki kod – mimo że na pierwszy, a nawet drugi rzut oka wydaje się poprawny – rzuci nam ładny błąd na konsolę. Czemu? Bo parser widzi go tak:

```
return; //bo niedbały programista zapomniał go tu dać
{}; //o, a co tu robi definicja obiektu?
```

Przez to małe głupstwo kiedyś zraziłem się do JSON i za wszelką cenę próbowałem go omijać. Dlatego na sam początek porad ot, taki mały kwiatek z własnego doświadczenia.

Krótszy zapis

```
var width = 0;
var height = 0;
var img = null;
var elem = null;
```

vs.

```
var width = 0
,height = 0
,img = null
,elem = null;
```

IMO czytelniej. Gdyby ktoś pytał czemu przecinek jest przerzucony na początek nowej linii, a nie zostawiony na końcu poprzedniej: spróbuj pousuwać parę zmiennych z listy. Jeśli przecinek jest na końcu linijki, musisz usunąć interesującą Cię zmienną, a następnie przecinek. Tak usuwasz jedno i drugie za jednym zamachem i na pewno nie zostawisz przecinka przez nieuwagę.

> Od kiedy ten tutorial powstał, jednak powróciłem do "normalnego" sposobu zapisu przecinków. Mój styl można podejrzeć **na moim GitHubie (https://github.com/Comandeer)**, a zwłaszcza **w projekcie BEMQuery (https://github.com/BEMQuery/bemquery-package-boilerplate)**.

Strict mode

To małe cudeńko bardzo ułatwia życie, utrudniając je. Włączenie tzw. "strict mode" (tryb ścisły jak ktoś polski lubi) usuwa najbardziej bugowate części JS (np. `with` czy też ciut naprawia stringi w `setTimeout`). Co więcej, nie pozwala nam tworzyć nieświadomie zmiennych globalnych, np tak:

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

W trybie ścisłym każda zmienna musi być zadeklarowana przed użyciem

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

A jak włączyć naszego małego przyjaciela? Prosto

```
"use strict"; //na samym początku skryptu żeby działał wszędzie
function a()
{
    "use strict"; //lub w funkcji, aby działał tylko w niej
}
```

Od razu uprzedzam, że jeśli nigdy nie zwracałeś uwagi na jakość swojego JS, to po włączeniu strict mode skrypty Ci się wykrzaczą. Ale przystosowanie ich do strict mode wcale nie jest trudne, a w przyszłości na pewno przyniesie korzyści (bo np. ociupinkę zmieni się składnia JS).

Literały

Spójrzmy na ten kod:

```
var a = new Array(1, 2, 3)
,b = new Object();

b.a = 1;
b.b = 2;

for(var i = 0; i<a.length; i++)
    console.log(a[i]);

for(var x in b)
    console.log(b[x]);
```

Wydaje się zupełnie poprawny, prawda? Owszem, może i jest poprawny, ale nie bezpieczny!

```
Array = function() {return '';};
Object = Array;

var a = new Array(1,2,3)
,b = new Object();

b.a = 1;
b.b = 2;

for(var i = 0; i<a.length; i++)
    console.log(a[i]);

for(var x in b)
    console.log(b[x]);
```

I mamy krzak, nawet w strict mode! Bezpieczniej użyć literałów:

```
Array = function() {return '';};
Object = Array;

var a = [1,2,3]
,b = {a:1, b:2};

for(var i = 0; i < a.length; i++)
    console.log(a[i]);

for(var x in b)
    console.log(b[x]);
```

Tych wykrzaczyć się nie da ;)

setTimeout

Nie przekazuj nazwy funkcji jako stringa!

```
setTimeout("funkcja()", 1000); //don't do this!
```

Tym samym wywołujesz sobie `eval`, a jak każdy wie – `eval` jest [s]złe[/s] niepotrzebnie wykorzystywane, co jedynie obniża wydajność! Przekaż uchwyt do funkcji:

```
setTimeout(funkcja, 1000);
```

A jak już musisz parametry przekazać:

```
setTimeout(function(){funkcja(1, 2);}, 1000);
//lub
setTimeout(funkcja, 1000, 1, 2);
```

Pętle i obiekty

Masz obiekt i musisz po nim poiterować? Zapewne robisz coś takiego:

```
var o = {
    a: 1
    ,b: 2
    ,c: 3
};

for(var x in o)
{
    console.log(o[x]);
}
```

Zgadłem? No to źle robisz:

```
Object.prototype.oops = 'BUGAHA!';

var o = {
    a: 1
    ,b: 2
    ,c: 3
};

for(var x in o)
{
    console.log(o[x]);
}
```

Powyższy kod wyświetli nam także 'BUGAHA!' (bo `for..in` iteruje także po wszystkich nienatywnych rozszerzeniach prototypu `Object`). Nie tego chcemy, prawda? A wystarczy dodać jedną linijkę:

```
Object.prototype.oops = 'BUGAHA!';

var o = {
    a: 1
    ,b: 2
    ,c: 3
};

for(var x in o)
{
    if(o.hasOwnProperty(x))
        console.log(o[x]);
}
```

I już. Metoda `hasOwnProperty` sprawdza czy wartość podana jako x na pewno jest częścią naszego obiektu i czy nie pochodzi z prototypu.

Jest też inny sposób, aby zupełnie ominąć jakiekolwiek prototypy i nie martwić się o nie:

```
Object.prototype.oops = 'BUGAHA!';

var o = Object.create(null);
o.a = 1;
o.b = 2;
o.c = 3;

for(var x in o)
{
    console.log(o[x]);
}
```

`Object.create` tworzy nam obiekt z prototypu podanego jako pierwszy parametr, tak więc tworzymy obiekt z pustym prototypem (domyślnie jest to `Object.prototype`). W starszych przeglądarkach ten sposób nie działa.

Jeszcze ładniej można to zrobić, korzystając z `Object.keys` (która to metoda nie szuka niczego w prototypach i zwraca wszystko w postaci normalnej tablicy kluczy):

```
var o = {
    a: 1
    ,b: 2
    ,c: 3
};

Object.keys(o).forEach(function(x)
{
    console.log(o[x]);
});
```

Funkcje natychmiastowego wywołania i przestrzenie nazw

Każdy doskonale wie, że zmienne globalne są bleeee. Jednak w wielu skryptach można znaleźć coś takiego:

```
var width = 0;
var height = 0;
var img = null;
var elem = null;
//itp.
```

Tym sposobem brudzimy sobie globalny scope!

```
console.log(window['width']);
```

A można lepiej, wykorzystując zasięg zmiennych:

```
(function()
{
    var width = 0;
    var height = 0;
    var img = null;
    var elem = null;
}());
console.log(window['width']);
```

OK, a jeśli chcemy coś specjalnie umieścić w globalnym scope, np. funkcje naszego super-hiper API? Oczywiście głupim pomysłem jest ładowanie oddzielnie wszystkich 150+ funkcji, bo istnieje szansa, że coś naszego nadpisze funkcje już używane na stronie (np. funkcja o nazwie `resizeImg`). Wtedy możemy posłużyć się przestrzenią nazw:

```
var API = {
    resizeImg: function()
    {
        console.log('wywołano');
    }
};
API.resizeImg();
```

Po połączeniu obydwu metod możemy osiągnąć coś takiego

```
(function($)
{
    var API = {}
    ,resizeImg = jakiswarunek ? function() {console.log('a');} : functio
n() {console.log(b);};

    API.resizeImg = resizeImg;
    $.API = API;
}(window))
```

Voila! W globalnym scope mamy tylko to, co chcieliśmy mieć!

Feature detection

> Sniffing an user agent is like sniffing a glue
> *— porneL*

Dlatego też, zamiast opierać się na wątpliwym przekonaniu, że w IE 5.5.1733958399 zainstalowanym pod Win XP z SP 14 na pewno to działa, warto sprawdzić czy naprawdę funkcja x istnieje i jest funkcją. Wyobraźmy sobie choćby, że chcemy stworzyć obiekt przy pomocy `Object.create`, ale nie jesteśmy pewni czy ta metoda istnieje:

```
var o = Object.create(null);
```

W starszych IE wywali nam ładny ReferrenceError. A wystarczy sprawdzić czy ta metoda istnieje:

```
if(typeof Object.create === 'function')
```

Jeśli nie, to można walnąć polyfilla (https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/create#Polyfill). Na tej samej zasadzie działa cała biblioteka Modernizr (http://modernizr.com).

Event delegation

Zdarzenia bąbelkują (jak ktoś nie wierzy, to niech poczyta (http://www.quirksmode.org/js/events_order.html)). Dlatego też możliwe staje się odkrycie, że jakiś tam akapit w jakimś tam divie został kliknięty.

```
(function(d)
{
    d.getElementsByTagName('div')[0].addEventListener('click', function(
e)
    {
        var t = e.target;
        if(t.tagName.toLowerCase() === 'p')
            console.log(t.innerHTML);
    }, false);
}(document));
```

Kiedy warto tego użyć? Jeśli np. mamy dużo przycisków na stronie i wszystkie są w jednym rodzicu. Zamiast przypisywać zdarzenia do każdego z nich, można przypisać te zdarzenia do ich rodzica i za pomocą `e.target` (w IE `e.srcElement`) sprawdzić, co tak naprawdę zostało kliknięte. Przydaje się także przy stronach ajaksowych, gdzie część elementów interaktywnych zostanie dodana po wczytaniu strony. Wtedy można doczepić zdarzenie np. do `body` i mieć pewność, że każdy przycisk będzie klikalny.

Copyright © by Comandeer (https://www.comandeer.pl).