Table of Contents *generated with DocToc*

JavaScript

autor: Bartosz Cytrowski

Instrukcje i wyrażenia

Język JavaScript składa się z dwóch rodzajów konstrukcji - są to:

- instrukcje (`statements`), np. `var x`
- oraz wyrażenia (`expressions`), np. `1 + 2`

Wyrażenia mogą składać się z innych wyrażeń. Każde wyrażenie posiada swoją wartość. Wartość wyrażenia zawsze posiada typ, np.

```
1 // Formalnie rzecz biorąc wartość liczbowa `1` może być traktowana jak wyrażenie o wartości liczbowej `1`
1 + 2 // Wyrażenie o wartości `3`, wkładające się z operatora dodawania (znak `+`) i dwóch wyrażeń o wartościach liczbowych: `1` i `2`
```

Typy danych

Typy danych w JavaSkrypcie dzielimy na proste i złożone.

Do typów prostych zaliczamy:

- `Boolean` - wartości logiczne `true` oraz `false`

- `Null` - wartość `null`
- `Undefined` - wartość `undefined`
- `Number` - liczby, np. `1`, `0.2`, `.2`
- `String` - tekst, np. `"foo"`, `'fizz buzz'`, `` `test` ``
- `Symbol` (ten typ pojawił się dopiero w ES6 - ECMAScript 2015, będę o nim pisał później)

Żeby sprawdzić jakiego typu jest wartość należy użyć operatora `typeof`, np.

```
typeof 9 // -> 'number'
```

Warto zapamiętać, że operator `typeof` zwraca wartość typu `String` (ciąg znaków).

Należy zwrócić uwagę na to, że wartości typu `String` jako jedyne zapisane są w cudzysłowiu: `'` (formalnie rzecz biorąc jest to znak oznaczający minutę, ale utarło się, że w językach programowania używamy go jako cudzysłowia). Można używać również `"` (sekundy). Ważne, żeby zamykać dany tekst tym samym znakiem, którym go otwieramy.

W JavaSkrypcie popularaniejszym znakiem otwierającym tekst jest `'` - głównie ze względu na to, że w `HTML` częściej używamy `"` i stwarzałoby to problem przy składaniu tekstu, np.

```
'<h1 id="main">Test</h1>' // poprawny ciąg znaków
"<p class=\"article\">Some text</p>" // dozwolone jest używanie znaku
ucieczki `\` w celu zmiany znaczenia znaku `"` - w tym wypadku tekst jest
składany poprawnie, ale stwarza to problemy w czytaniu
 "<h2 id="secondary">More text</h2>" // tutaj mamy błąd składni -
interpreter widzi tekst "<h2 id=", a po nim symbol `secondary`, który nie
został nigdzie zdefiniowany z użyciem słowa kluczowego `var` - nie ma to
sensu i skończy się błędem
```

W ES6 doszedł nam jeszcze jeden znak rozpoznawany jako początek/koniec tekstu: `` ` `` (backtick; grave accent; na klawiaturze znajduje się w okolicach tyldy: `~`) - pozwala on na osadzanie wyrażeń w obrębie tekstu, np.

```
`Wynikiem sumy liczb ${10} i ${20} jest ${10 + 20}`
// "Wynikiem sumy liczb 10 i 20 jest 30"
```

Ten sam rezultat uzyskalibyśmy używając znaku `+`, który w przypadku tekstu oznacza operację `konkatenacji` (łączenia ciągów znaków)

```
'Wynikie sumy liczb ' + 10 + ' i ' + 20 + ' jest ' + (10 + 20)
// w zmiennej `z2` znajduje się taki sam tekst jak powyżej
```

Warto zwrócić uwagę na występienie nawiasów okrągłych - grupują one niezależne części wyrażenia i wskazują interpreterowi kolejność obliczania wartości. Gdyby przykład ten zapisać bez nawiasów, to uzyskamy inny efekt:

```
'Wynikie sumy liczb ' + 10 + ' i ' + 20 + ' jest ' + 10 + 20
// "Wynikiem sumy liczb 10 i 20 jest 1020"
```

Wynika to z faktu, że interpreter dokonuje tzw. koercji typów (`coertion`) i gdy po jednej stronie znaku `+` ma liczbę, a po drugiej ciąg znaków, to jedną i drugą wartość sprowadza do ciagu znaków. Liczba `10` staje się ciągiem znaków `"10"` i następuje konkatenacja. Podobnie z liczbą `20`. Różne typy danych ulegają koercji w różny sposób. Warto mieć do na uwadze przy budowaniu ciągu znaków z wyników innych wyrażeń.

Drugą kategorią typów są typy złożone - w ogólności `Object`, a w szególności `Array`, `Function` oraz cała masa obiektów wbudowanych w JavaScript. Poświęcę im osobny rozdział po tym, jak omówimy zmienne. Najważniejsze różnice opiszę w trakcie omawiania prototypów.

Zmienne

Pod pojęciem zmiennej rozumiemy symbol, któremu nadejemy nazwę oraz określamy, jaką wartość przechowuje.

Zmienne deklarujemy z użyciem słowa kluczowego `var`, np.

```
var x // instrukcja deklaracji zmiennej o nazwie `x`
```

Żeby przypisać wartość do zadeklarowanej zmiennej używamy znaku `=`, czyli operatora przypisania (`assignment`)

```
var x // instrukcja deklaracji zmiennej o nazwie `x`
x = 10 // instrukcja przypisania wartości `10` do zmiennej o nazwie `x`
```

Zapis skrócony wygląda tak:

```
var x = 10 // instrukcja deklaracji zmiennej o nazwie `x` i wartości `10`,
nazywana też definicją zmiennej lub inicjalizacją zmiennej
```

Do zmiennych wpisujemy zwykle wartości różnych wyrażeń, np.

```
var x = 1 + 2
```

W powyższym zapisie interpreter JavaSkryptu w pierwszej kolejności obliczy wartość wyrażenia `1 + 2`, a następnie pod zmienną `x` podstawi `3`.

Wartość przechowywaną w zmiennej można zmieniać, np.

```

```

```
var x = 10
x = 12
x = 15
```

Lub wyliczać na podstawie wartości przechowywanych w innych zmiennych, np.

```
var x = 10
var y = 20
var z = x + y // -> tutaj do zmiennej `z` wpisujemy wartość `30`
```

W formie skróconej:

```
var x = 10, y = 20, z = x + y
```

Dopuszczalny jest też zapis:

```
var x = y = 20
```

W tym przypadku interpreter najpierw przypisze wartość `20` do zmiennej `y`, a później wartość zmiennej `y` do zmiennej `x`, co da nam wartość `20` w jednej i drugiej.

W ES6 (EcmaScript 6; nowszy standard JavaScript) pojawiło się rozróżnienie na symbole, które nie mogą zmieniać wartości oraz te, które mogą:

```
let x = 10 // zmienna
x = 12

const y = 10 // stała
y = 12 // ERROR! Nie można zmienić wartości trzymanej pod symbolem y,
ponieważ został on zadeklarowany z użyciem słowa kluczowego `const`
```

Atrybuty obiektów

Innym elementem pozwalającym na przechowywanie wartości są tzw. atrybuty/pola/właściwości/propercje obiektów. Możemy z nich skorzystać, gdy tworzymy obiekt. Obiekt służy do zgrupowania kilku wartości, które są ze sobą powiązane w pewien logiczny sposób, np.

```
var position = { x: 10, y: 20 }
```

W powyższym zapisie nawiasy klamrowe (zwane też "wąsatymi") służą stworzeniu obiektu za pomocą tzw. literału (`object literal`). Gdy chcemy odwołać się do wartości przechowywanej w atrybucie obiektu musimy odwołać się do symbolu, który na ten obiekt wskazuje, a następnie do nazwy atrybutu, np.

```
position.x // -> to daje nam wartość `10`
```

Atrybuty do obiektów można dodawać w sposób dynamiczny, tzn. obiekt nie musi posiadać ich od momentu, gdy go utworzymy - można dodać je później, np.

```
position.z = 30
```

Dozwolone jest utworzenie pustego obiektu:

```
var foo = {}
```

Znak `.` jest operatorem, który pozwala nam odwoływać się do atrybutów obiektu. Innym sposobem na wykonanie tych samych operacji jest stosowanie zapisu z nawiasami kwadratowymi (`square brackets`), np.

```
var position = { x: 10, y: 20 }
position['x'] // -> to daje nam wartość `10`
position['z'] = 30 // -> to sprawia, że do obiektu zostaje dodane nowe
pole o nazwie `z` i wartości `30`
```

Zwróćmy uwagę na fakt, że nazwa `x` jest zapisana w cudzysłowiu - to oznacza, że jest ciągiem znaków. Pamiętajmy, że ciągi znaków można budować wyrażeniami.

Zapis z nawiasami kwadratowymi pozwala nam na realizację dynamicznego dostępu do atrybutów obiektu (`dynamic property access`), gdy np. w czasie pisania kodu nie wiemy jeszcze, z którego atrybutu będziemy chcieli odczytać wartość lub do którego ją wpisać (zależy to od interakcji z użytkownikiem), np.

```
var nameOfAttribute = 'red'
var options = {
  red: '#f00',
  green: '#0f0',
  blue: '#00f'
}
options[nameOfAttribute] // otrzymamy tutaj wartość '#f00'
```

Gdy założymy, że wartość zmiennej `nameOfAttribute` będzie zależała np. od tego, co użytkownik wpisze w formularzu na stronie, mechanizm ten daje nam bardzo dużo możliwości uproszczenia kodu.

Warto też mieć na uwadze, że atrybuty obiektów mogą przechowywać inne obiekty, co pozwala nam opisywać dużo bardziej złożone struktury danych, np.

```
var player = {
  name: 'Mario',
  position: {
    x: 10,
    y: 20
  },
  coinsCollected: 30
}
```

Obiekty `Array`

Przykładowe zastosowanie obiektu poznaliśmy przy omawianiu zastosowania jego atrybutów do przechowywania wartości. Obiekty `Array` pozwalają nam z kolei na wygodne tworzenie kolekcji wartości, np.

```
var numbers = [1, 2, 3]
```

Nawiasy kwadratowe w tym przypadku oznaczają literał tworzący obiekt `Array`, zwany też tablicą lub macierzą jednowymiarową. W wielu językach programowania tablice mogą zawierać wartości tylko i wyłącznie jednego typu. W JavaSkrypcie nie ma tego ograniczenia, np.

```
var items = [1, 'foo', { x: 10 }] // ten zapis jest jak najbardziej
poprawny, aczkolwiek niezalecany, ze względu na utrudnioną późniejszą
analizę działania kodu
```

Zwróćmy uwagę, że w przypadku obiektu `Array` nie musimy jawnie określać nazw jego atrybutów. Zapis:

```
var vowels = ['a', 'e', 'i', 'o', 'u']
```

oznacza w pewnym uproszczeniu:

```
var vowels = {
  0: 'a',
  1: 'e',
  2: 'i',
  3: 'o',
  4: 'u',
  length: 5
}
```

Warto zauważyć, że atrybut `length` został ustawiony automatycznie na podstawie zawartości tablicy. Jako, że obiekty `Array` są obiektami, aplikują się do nich te same zasady:

```
var letter = vowels[4] // -> do zmiennej `letter` wpisujemy wartość `u`
vowels[5] = 'y' // -> sprawi, że zmienna `letter` będzie zawierała tablicę
['a', 'e', 'i', 'o', 'u', 'y']
vowels.length // -> dostaniemy tutaj wartość 6, ponieważ operacja
przypisania wartości `y` powiększyła nam tablicę o 1 element
```

Należy uważać z tym sposobem dodawania elementów do tablicy, ponieważ, gdybyśmy alternatywnie wykonali:

```
vowels[6] = 'y'
```

to na pozycji 5 w tablicy zostałaby wstawiona wartość `undefined`, a próba odczytania `vowels.length` zwróciłaby wartość 7, a nie 6. Otrzymalibyśmy tzw. tablicę rozrzedzoną (`sparse array`). Całe szczęście obiekty `Array` posiadają całą gamę funkcji, które pozwalają nam uniknąć tego typu pomyłek - będę o nich pisał później.

W obiektach `Array` nie możemy stosować do odczytu wartości na konkretnej pozycji operatora kropki. Wynika to z faktu, że `vowels.5` jest uznawany przez interpreter JavaSkryptu za błąd składni. JavaScript rozumie ten zapis jak `vowels` oraz `.5` czyli nazwę zmiennej oraz liczbę `0.5`. Należy o tym pamiętać również w przypadku innych obiektów, jeżeli nazwy ich atrybutów są liczbami, np.

```
var x = {
  6: 50,
  foo: 10
}
x['foo'] // -> daje wartość `10`
x.foo // -> daje wartość `10`
x['6'] // -> daje wartość `50`
x.6 // ERROR
```

Ponadto nazwy atrybutów obiektu mogą również zawierać tzw. białe znaki np. `spację`. W takim wypadku należy również uważać przy ich wstawianiu w definicji obiektu, np.

```
var x = {
  'be awesome': true
}
```

naturalnie zapis:

```
var x = {
  be awesome: true
```

```
}
```

zostanie uznany przez interpreter za błąd składni.

W ES6 pojawiła się skrócona możliwość dynamicznego dodawania atrybutów obiektu:

```
var name = 'foo'
var x = {
  [name]: 10
}
x.foo // -> to da nam wartość `10`
```

Ten zapis jest równoznaczny z:

```
var name = 'foo'
var x = {}
x[name] = 10
x.foo // -> to da nam wartość `10`
```

Obiekty `Function`

Funkcje w JavaSkrypcie możemy definiować na dwa sposoby. Pierwszy i najbardziej uniwersalny nazywany jest deklaracją funkcji `function declaration`, np.

```
function foo(a, b, c) {

}
```

Deklaracja funkcji składa się:

- ze słowa kluczowego `function`,
- nazwy funkcji,
- po której następują nawiasy okrągłe, grupujące nazwy argumentów wywołania, oddzielone przecinkami
- oraz nawiasów klamrowych oznaczających ciało funkcji

Kod zapisany w ciele funkcji nie zostanie uruchomiony dopóki jej nie wywołamy. W celu wywołania funkcji należy po symbolu przechowującym jej definicję użyć nawiasów okrągłych (`curly braces`), np.

```
function sum(a, b) {
  return a + b
}
sum(1, 2) // daje nam wartość `3`
```

Wywołanie funkcji jest wyrażeniem. Wynikiem tego wyrażenia jest wartość, którą umieścimy w obrębie ciała funkcji po słowie kluczowym `return`. Jeżeli kod ciała funkcji w trakcie wywołania nie dotrze do linijki z `return` to funkcja zwróci wartość `undefined`, np.

```
function sum(a, b) {
  a + b // tutaj przez nieuwagę pomijamy `return`
}
sum(1, 2) // daje nam wartość `undefined`
```

Jest to błąd popełniany dosyć często przez początkujących programistów JavaSkryptu.

Drugim sposobem definiowania funkcji jest tzw. `function expression`, czyli przypisanie do zmiennej funkcji anionimowej (nie posiadającej nazwy), np.

```
var buzz = function () {
  return 10
}
```

Funkcję w tej konstrukcji można też nazwać, ale nazwa ta zostanie w pewnym sensie zignorowana - nie praktykuje się tego zapisu, bo wprowadza on zamieszanie:

```
var buzz = function foo() { // DO NOT DO IT AT HOME!
  return 10
}
buzz() // zwraca wartość `10`
foo() // ERROR - foo is not a function
```

`Function expression` jest jedynym sposobem na użycie trzeciego wariantu definicji funkcji: `arrow function` (funkcja strzałkowa; fat-arrow function), np.

```
const sum = (a, b) => a + b
// gdy mamy więcej niż jeden argument wywołania funkcji to musimy je
zgrupować nawiasami okrągłymi
// jeżeli po strzałce znajduje się wyrażenie, to return nie jest potrzebny

const inc = a => a + 1
// jezeli mamy jeden argument to nie potrzebujemy nawiasów okrągłych

const sum2 = (a, b) => {
  let result = 0
  result += a // skrót od: result = result + a
  result += b // skrót od: result = result + b
  return result
}
// jeżeli wykonujemy kilka instrukcji po strzałce w celu wyliczenia
```

```
wartości zwracajen przez funkcję, to niezbędne są nawiasy klamrowe
oznaczające blok ciała funkcji – wymagane jest też słowo kluczowe `return`
jeżeli chcemy rzeczywiście zwrócić z funkcji wartość
```

Zapis ten omówimy dokładnie przy poruszaniu tematyki kontekstu wywołania funkcji - czyli znaczenia złowa kluczowego `this`.

Dzięki obiektom `Function` funkcje w JavaScript są traktowane jako tzw. `first class citizen`. W wielu językach programowania obiektowego funkcje nie występują jako niezależne wartości. W JavaSkrypcie możemy przypisywać je do zmiennych oraz przekazywać jako wartości argumentów wywołania funkcji.

Wartości i referencje

Podstawową cechą odróżniającą typy złożone od typów prostych jest to, że wartości typów prostych są przekazywane "przez wartość" a typów złożonych - "przez referencję".

Żeby wyjaśnić różnice między wartościami a referencjami musimy najpierw zrozumieć działanie tzw. operatorów logicznych.

```
var x = 9
var y = 9

x === y // – daje nam wartość logiczną `true`
```

```
var x = { a: 10 }
var y = { a: 10 }

x === y // – daje nam wartość logiczną `false`
```

Na pierwszy rzut oka wydaje się to nielogiczne, ale trzeba pamiętać, że `object literal` tworzy za każdym razem nowy obiekt. Operator `===` służy do wykonywania tzw. `strict equality comparison` i sprawdza czy mamy do czynienia z dokładnie tym samym obiektem, a nie takim samym - nie porównuje zawartości obiektu, a referencję. Ważne jest to, że obiekty nigdy nie są przechowywane w zmiennych bezpośrednio. Można to sobie wyobrazić tak, że gdzieś w pamięci komputera ten obiekt składowany jest binarnie, ale to, co zapisujemy w zmiennej jest tylko adresem, pod którym znajdziemy rzeczywisty obiekt - nazywamy to referencją.

Co ciekawe, w JavaSkrypcie mamy jeszcze drugi operator porównania: `==`, ale i on nas w tym przypadku rozczaruje

```
var x = { a: 10 }
var y = { a: 10 }

x == y // – daje nam wartość logiczną `false`
```

Sytuacja się zmieni, jeżeli wartości zmiennych ustawimy w ten sposób:

```
var x = { a: 10 }
var y = x

x === y // - daje nam `true`
x == y // - też daje nam `true`
```

Referencje mogą też zaskoczyć w innym przypadku:

```
var x = { a: 10 } // do zmiennej `x` wpisujemy referencję na obiekt { a:
10 }
var y = x // do zmiennej `y` wpisujemy tę samą referencję

y.a = 20 // zmieniamy wartość atrybutu obiektu, na który wskazuje
referencja w zmiennej `y`

x.a === 10 // `false` - zmieniła się też wartość atrybutu obiektu, do
którego referencję trzymamy w zmiennej `x` - w końcu obie zmienne wskazują
na ten sam obiekt
x.a === 20 // `true` - tutaj nie ma niespodzianek
```

W przypadku typów prostych nie ma tego problemu:

```
var x = 10
var y = x

y = 20
x === 20 // false
x === 10 // true
y === 20 // true
```

Scope (widoczność)

W takcie pisania kodu istotne jest zwracanie uwagi na to, które symbole są dostępne w danym bloku kodu.

Jeżeli odwołamy się do symbolu, który nie został zadeklarowany i będziemy chcieli odczytać z niego wartość, to otrzymamy błąd, np.

```
var y = x // ERROR: x nie jest zdefiniowane
```

W sytuacji, gdy będziemy chcieli wpisać wartość pod symbol, który nie został wcześniej jawnie zadeklarowany słowem kluczowym `var`, to niejawnie utworzymy tzw. zmienną globalną (`global variable`), np.

```
function foo() {
  x = 10
}
foo() // to wywołanie sprawiło, że pod `window.x` znajduje się wartość
`10`
```

To, jakie symbole dostępne są w danym bloku kodu zależy od kilku czynników.

Hoisting

Symbole utworzone za pomocą słowa kluczowego `var` ulegają tzw. `hoistingowi` (windowaniu). Polega to na tym, że parser przed uruchomieniem kodu skanuje go w poszukiwaniu instrukcji `var [nazwa zmiennej]`, np. `var x`, tworząc spis dostępnych symboli. Dzięki temu mechanizmowi poniższy kod:

```
x = 22
var x
```

Zachowuje się tak, jakby był napisany w ten sposób:

```
var x
x = 22
```

Inny przykład, zahaczający o 2 sposoby definiowania funkcji:

```
var x = 20

var y = fizz()
var z = buzz() // ERROR

// function declaration
function fizz() {
  return 10
}

// function expression
var buzz = function () {
  return 15
}
```

będzie interpretowany jako:

```
var x, y, z, buzz;
// funkcja fizz została zdefiniowana z użyciem tzw. `function
declaration`, które ulega hoistingowi
```

```
function fizz() {
  return 10
}

x = 20
y = fizz() // pod symbol `y` zostanie wpisana wartość zwracana z funkcji
`fizz` za pomocą słowa kluczowego `return` - liczba 10
z = buzz() // ERROR: buzz w tym momencie posiada w sobie wartość
`undefined` - nie jest jeszcze funkcją, więc nie można użyć po nazwie
`buzz` nawiasów okrągłych (oznaczających wywołanie)

// `function expression` nie ulega hoistingowi
buzz = function () {
  return 15
}
```

Prototypy

Każdy obiekt w JavaSkrypcie posiada specjalny atrybut `__proto__`. Jego wartością zawsze jest referencja na inny obiekt lub wartość `null`. Nie odwołujemy się do niego bezpośrednio, ale jeżeli spróbujemy się odwołać do atrubutu, którego obiekt nie posiada, to interpreter odwoła się do obiektu wskazywanego przez `__proto__` i w nim będzie szukał żądanej wartości. Jeżeli tam jej nie znajdzie, to odwoła się do atrybutu `__proto__` obiektu, na który wskazuje atrybut `__proto__` itd. Jeżeli dotrze do `__proto__` równego `null` to przestaje szukać i zwraca `undefined`.

Atrybut `__proto__` można ustawić na kilka sposobów.

1. Jeżeli tworzymy obiekt z użyciem `{}` (object literal), to w `__proto__` tworzonego obiektu zapisywana jest referencja do `Object.prototype`. Sam `Object.prototype` z kolei w swoim `__proto__` posiada `null`.

2. Jeżeli tworzymy obiekt z użyciem `Object.create(x)`, to referencja do obiektu `x` umieszczana jest w atrybucie `__proto__` nowo tworzonego obiektu. W tym wypadku `x` może być obiektem lub wartością `null`. W innym wypadku otrzymamy błąd.

3. Jeżeli tworzymy obiekt z użyciem funkcji konstruktora, to w jego atrybucie `__proto__` wyląduje wartość atrybutu `prototype` tej funkcji. Warto pamiętać, że `prototype` domyślnie zawiera tylko informacje o konstruktorze i prototypie pochodzącym z `Object.prototype`

```
function Foo() {}

var x = new Foo()

// > Foo {}
//   __proto__:
//     constructor: ƒ Foo()
//     __proto__:
//       constructor: ƒ Object()
//       hasOwnProperty: ƒ hasOwnProperty()
//       isPrototypeOf: ƒ isPrototypeOf()
```

```
//       propertyIsEnumerable: ƒ propertyIsEnumerable()
//       toLocaleString: ƒ toLocaleString()
//       toString: ƒ toString()
//       valueOf: ƒ valueOf()
//       __defineGetter__: ƒ __defineGetter__()
//       __defineSetter__: ƒ __defineSetter__()
//       __lookupGetter__: ƒ __lookupGetter__()
//       __lookupSetter__: ƒ __lookupSetter__()
//       get __proto__: ƒ __proto__()
//       set __proto__: ƒ __proto__()
//       __proto__: null  - ten atrybut nie jest pokazywany w konsoli, ale
możemy go zobaczyć po wpisaniu `x.__proto__.__proto__.__proto__`
```

Przykład z rozbudowanym prototypem dostępnym w atrybucie `prototype` konstruktora:

```
function Foo() {}

var x = new Foo()

Foo.prototype.add = function (a, b) { return a + b }
Foo.prototype.sub = function (a, b) { return a - b }
// > Foo {}
//   __proto__:
//     constructor: ƒ Foo(),
//     add: ƒ (a, b), - te właściwości pochodzą z `prototype` funkcji
//     sub: f (a, b) - te właściwości pochodzą z `prototype` funkcji
//     __proto__:
//       constructor: ƒ Object()
//       hasOwnProperty: ƒ hasOwnProperty()
//       isPrototypeOf: ƒ isPrototypeOf()
//       propertyIsEnumerable: ƒ propertyIsEnumerable()
//       toLocaleString: ƒ toLocaleString()
//       toString: ƒ toString()
//       valueOf: ƒ valueOf()
//       __defineGetter__: ƒ __defineGetter__()
//       __defineSetter__: ƒ __defineSetter__()
//       __lookupGetter__: ƒ __lookupGetter__()
//       __lookupSetter__: ƒ __lookupSetter__()
//       get __proto__: ƒ __proto__()
//       set __proto__: ƒ __proto__()
//       __proto__: null  - ten atrybut nie jest pokazywany w konsoli, ale
możemy go zobaczyć po wpisaniu `x.__proto__.__proto__.__proto__`
```

Właściwość `constructor` pamięta referencję do funkcji, która służyła do utworzenia danego obiektu.

Kontekst wywołania funkcji

Kontekstem wywołania funkcji nazywamy wartość, na którą w trakcie wywołania funkcji wskazuje słowo luczowe `this`. Ta wartość zawsze jest obiektem. Podobnie jak wartości argumentów wywołania - wartość `this` określa się w momencie wywołania funkcji. Mamy kilka sposobów, aby to osiągnąć. Załóżmy, że mamy funkcję:

```
function foo() { return this }
```

Możemy wywołać ją bez jawnego określania wartości `this`:

```
foo() // -> zwraca nam obiekt `window`
```

Jeżeli w naszym skrypcie znajdzie się dyrektywa `"use strict"`:

```
foo() // -> zwraca wartość `undefined`
```

Wyjątkiem od reguły są funkcje utworzone za pomocą `Function.prototype.bind`:

```
var boo = foo.bind('Test')

boo() // -> zwraca `String('Test')`
```

> `Function.prototype.bind` zwraca nową funkcję, w której możemy zaszyć kontekst wywołania oraz wartości kolejnych argumentów.

Wyrażenie ze słowem kluczowym `new`:

```
function foo() { return this }

new foo() // zwraca pusty obiekt: `{}`, który w __proto__ posiada
foo.prototype, które z kolei w swoim __proto__ posiada Object.prototype
```

> `new` tworzy nowy pusty obiekt i przekazuje go do funkcji `foo` jako `this` na czas jej wywołania. Gdy funkcja skończy działanie, `new` odbiera zmodyfikowany przez nią obiekt i zwraca jako wartość wyrażenia

> Wywołanie z `new` ignoruje wartość `this` ustawioną za pomocą `Function.prototype.bind`

Operator kropki `.`

```
var rectangle = {
  width: 10,
  height: 20,
  getArea: function() {
    return this.width * this.height
  }
}
```

```
rectangle.getArea() // -> zwraca nam wartość `200`
```

> Wywołanie funkcji, której nazwa jest poprzedzona kropką sprawia, że wartość na lewo od kropki staje się wartością `this` na czas wywołania tej funkcji. Oczywiście można to zrobić tylko i wyłącznie wtedy, gdy wartością po lewej jest obiekt i posiada on atrybut o nazwie na prawo od kropki, np.
>
> ```
> var y = { a: 10, blah: function () { return this }}
> y.blah() // -> to jest spoko i zwróci nam { a: 10, blah: f } (f
> oznacza tutaj funkcję anonimową)
>
> var x = { a: 10 }
> x.blah() // -> to wyrzuci błąd `blah is not a function`
> ```

Array.prototype.call oraz Array.prototype.apply

Wymienione funkcje (`call` oraz `apply`) w odróżnieniu od `Array.prototype.bind` nie zwracają nowej funkcji. Możemy ich jednak użyć, gdy potrzebujemy wywołać jakąś funkcję, jednoznacznie określając wartość `this`, z której funkcja ta ma skorzystać, np.

```
var x = {
  a: 10,
  getA: function () { return this.a }
}

x.getA() // -> zwraca `10`

var y = {
  a: 20
  // zwróćmy uwagę, że tutaj nie ma metody (atrybutu obiektu, którego
wartością jest funkcja) `getA`
}

x.getA.call(y) // -> zwraca `20`
x.getA.apply(y) // -> zwraca `20`
```

> Dzięki tej konstrukcji możemy "pożyczyć" sobie na chwilę metodę z jednego obiektu na potrzeby wykonania operacji na innym.

Inny przykład:

```
var x = {
  a: 10,
  getA: function () { return this.a }
}
```

```
var foo = x.getA // do zmiennej `foo` wpisujemy referencję do funkcji
trzymanej w atrybucie `getA` w obiekcie `x`

foo() // -> otrzymamy tutaj `undefined`, ponieważ wywołujemy funkcję bez
wskazania na wartość, którą ma być `this` – interpreter podstawia więc
obiekt `window`, a tak się składa, że `window.a` nie ma wartości, więc
dostajemy `undefined`; gdybyśmy używali na początku dyrektywy "use strict"
(co jest bardzo dobrą praktyką), to interpreter wyrzuciłby błąd mówiący,
że nie możemy skorzystać z atrybutu `a` wartości, która jest `undefined`

foo.call(x) // -> tutaj otrzymamy `10`
foo.apply(x) // -> tutaj też
```

> Metody `call` oraz `apply` wykonują to samo zadanie, ale różnią się sposobem wywołania - `call` przyjmuje dowolną liczbę argumentów, a `apply` tylko dwa - pierwszym jest wartość, którą chcemy podstawić pod `this`, a drugim tablica, której elementy zostaną użyte jako kolejne argumenty wywołania danej funkcji, np.
>
> ```
> function sum(a, b) { return a + b }
>
> sum.call(null, 1, 2) // -> this nie jest istotny w tym wypadku, więc
> używamy `null`; `1` ląduje pod argumentem `a`; `2` pod argumentem
> `b`; dostajemy wartość `3`
>
> sum.apply(null, [1, 2]) // -> tutaj też dostaniemy `3`
> ```

Arrow function

W ES6 pojawił się nowy sposób definiowania funkcji, które jest mocno zbliżony do tworzenia funkcji z użyciem `Array.prototype.bind`. Funkcje zdefiniowane za pomocą tzw. `arrow function` (czasami zwanego `fat arrow function`) używają takiej wartości `this`, którą widzą w nadrzędnym `scope`. Wynika z tego, że `this` jest dla nich określany w momencie definiowania a nie wywołania, np.

```
function foo() {
  var buzz = () => this

  return buzz()
}

foo.call('Test') // zwróci `String('Test')`
```

natomiast:

```
function foo() {
  var buzz = function () { return this }
```

```
  return buzz()
}

foo.call('Test') // zwróci nam `window`
```

Warto tutaj zaznaczyć, że przy tworzeniu obiektu za pomocą `object literal` samo `arrow function` może nas zaskoczyć:

```
var x = {
  a: 10,
  getA: () => this.a
}

x.getA() // zwróci nam `undefined`, ponieważ w momencie, gdy
definiowaliśmy wartość atrybutu `getA` thisem był obiekt `window`

// w przypadku "use strict" kod wysypałby się błędem mówiącym, że nie
możemy odczytać atrybutu `a` z `undefined`
```

Podsumowanie

Celem tego dokumentu jest zgromadzenie w jednym miejscu najczęściej spotykanych na początku przygody z JS schamatów związanych z ES5 oraz ES6. Brakuje tu wielu elementów, ale sukcesywnie będę je dodawał w miarę zapotrzebowania ze strony grupy 😃

Powodzenia!