

Wybrane elementy praktyki projektowania oprogramowania

Wykład 03/15

JavaScript, podstawy języka (2)

Wiktor Zychla 2021/2022

Spis treści

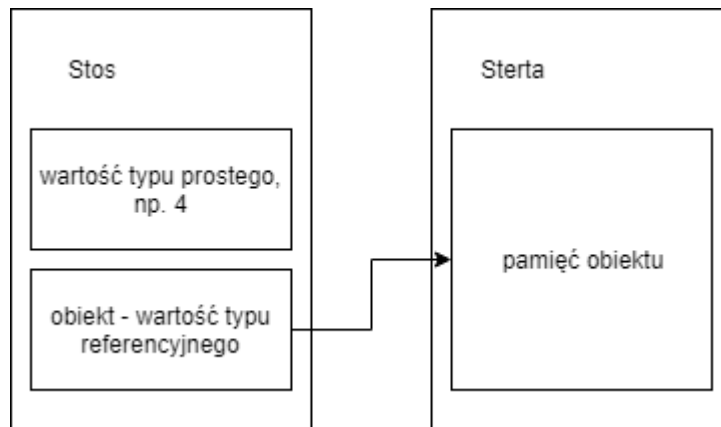
Podstawy języka, ciąg dalszy	2
Typy proste a referencyjne	2
Składnia/styl	2
String	3
Date	4
getter/setter – implementacja właściwości ze skutkami ubocznymi	4
Tablice	6
Wyjątki	7
Dialekty zawężające	7
JSON	7
Funkcje jako obiekty pierwszoklasowe	7

Podstawy języka, ciąg dalszy ...

Typy proste a referencyjne

Wartości typów prostych są reprezentowane w pamięci jako tablice bajtów, bezpośrednio w ramach stosu aktualnie wykonującej się funkcji.

Wartości typów referencyjnych są reprezentowane jako „referencje” do pamięci zawierającej stany obiektów, zarezerwowanej na sterce (czyli możliwej do współdzielenia między funkcjami).



Javascript ma, jak wiele języków (C/C++/Java/C# itd.), domyślną konwencję przekazywania do funkcji **przez wartość**, co w przypadku typu prostego oznacza kopię wartości, w przypadku referencji – kopię wartości referencji. W szczególności oznacza to że wewnątrz funkcji **nie może** zmienić wartości/referencji tak żeby zmiana była widoczna w miejscu wywołania.

```
function change(n) {  
    n = 2;  
    console.log(`po zmianie lokalnie w funkcji ${n}`, );  
}  
  
var n = 1;  
console.log(n);  
change(n);  
console.log(` w miejscu wywołania ${n}`);
```

Składnia/styl

Temat składni języka pominiemy stwierdzeniem, że w zakresie podstawowych konstrukcji imperatywnych (if/for/while/switch) Javascript zbyt przypomina inne znane już nam języki żeby było warto te tematy dodatkowo omawiać. Warto natomiast wspomnieć o wynikającym z wbudowanej w parser regule [Automatic Semicolon Insertion](#) (ASI) preferowanym stylu:

```
function f() {  
  
}
```

```
// czy

function g()
{
}

```

Istnieje możliwość użycia zewnętrznego narzędzia typu [linter](#), dla JS byłby to na przykład [ESLint](#), który może być zainstalowany jako dodatek do VS Code. Linter jest konfigurowany poleceniem

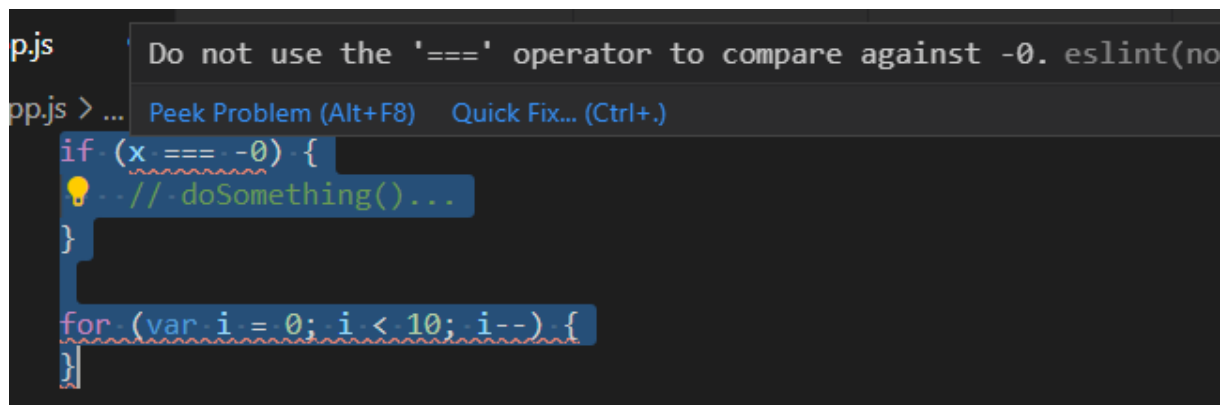
eslint –init

(uwaga na możliwy [problem w Windows](#))

i tworzy plik konfiguracyjny w którym można włączać/wyłączać wszystkie/poszczególne reguły

```
{
  "env": {
    "browser": true,
    "commonjs": true,
    "es2021": true
  },
  // "extends": "eslint:recommended",
  "parserOptions": {
    "ecmaVersion": 12
  },
  "rules": {
  }
}
```

Przykładowa walidacja



```
p.js
pp.js > ... Do not use the '===' operator to compare against -0. eslint(no
Peek Problem (Alt+F8) Quick Fix... (Ctrl+.)
if (x === -0) {
  // doSomething()...
}
for (var i = 0; i < 10; i--) {
}
```

String

Ogranicznikami napisów są tradycyjnie " i '". W dialekcie ES6 dodano ` jako ogranicznik szablonów (niżej).

Jeśli chodzi o same napisy, to warto zapoznać się z biblioteką standardową i znać podstawowe metody napisów (m.in. toLower/toUpper/replace/trim/indexOf/startsWith/endsWith).

Do formatowania zawartości napisów w Javascript używa się rozszerzenia języka dodającego mechanizm szablonów do literałów (tzw. [string templates](#)):

```
var n = 1, m = 2;
var string = `szablon z wartościami ${n}, ${m} i ${n+m}`;

console.log( string );
```

Date

Javascript w bibliotece standardowej posiada wzorowany na Javie interfejs operacji na datach. W tym przypadku również warto przestudiować dokumentację dostępnych metod oraz zapoznać się z tym jak określono standard [ISO 8601](#) i jak konwertować daty w javascript z/na taki właśnie format.

Użycie jedynie takiego formatu przy wymianie danych gwarantuje interoperacyjność z innymi technologiami.



Rysunek 1 Za <https://xkcd.com/1179/>

getter/setter – implementacja właściwości ze skutkami ubocznymi

Na wzór wielu języków programowania, JavaScript posiada lukier syntaktyczny do definiowania właściwości (properties), wymagający określenia akcesorów [get](#) i [set](#).

```
var foo = {
  _i : 0,
  get bar() {
    return foo._i;
  },
  set bar(i) {
    foo._i = i;
  }
}
```

```
}

console.log( foo.bar );

foo.bar = 5;

console.log( foo.bar );
```

Możliwe jest ogólniejsze podejście, użycie **Object.defineProperty** i tak zwanego [Property Descriptora](#), za pomocą którego można dodać do obiektu dowolną wartość (pole, właściwość, funkcję):

```
var foo = {
  _i : 0,
  get bar() {
    return foo._i;
  },
  set bar(i) {
    foo._i = i;
  }
}

Object.defineProperty( foo, 'qux', {
  get : function() {
    return 17;
  }
});

Object.defineProperty( foo, 'baz', {
  value : function() {
    return 34;
  }
});

console.log( foo.qux );
console.log( foo.baz() );
```

O ile do istniejącego obiektu można dodawać pola/metody za pomocą zwykłej składni (operatora `.` lub `[]`) o tyle dodanie do istniejącego obiektu właściwości (get/set) jest możliwe tylko w wyżej zademonstrowany sposób.

W trakcie wykładu powiemy o dodatkowych atrybutach descriptora (**writable**, **enumerable**, **configurable**).

Tablice

Tablice to obiekty o indeksach numerycznych, zoptymalizowane pod kątem szybkości dostępu oraz zużycia pamięci. W przeciwieństwie do wielu języków, Javascript nigdy nie wyrzuca wyjątków typu „out of bounds” ponieważ dostęp do komórki tablicy która nie posiada wartości zwraca wartość **undefined**.

Z uwagi na omówione wcześniej dynamiczne konwersje między typami, mylące może być więc zastosowanie idiomatycznej konstrukcji dla klauzuli **if**:

```
var a = [];  
  
a[100] = 1;  
  
if ( a[100] ) {  
    console.log( 'jest element' );  
} else {  
    console.log( 'nie ma elementu' );  
}
```

ponieważ takie podejście niepoprawnie rozpozna sytuację gdy elementem tablicy o wskazanym indeksie jest cokolwiek konwertowalnego do **false**:

```
var a = [];  
  
a[100] = 0;  
  
if ( a[100] ) {  
    console.log( 'jest element' );  
} else {  
    console.log( 'nie ma elementu' );  
}
```

Jest to jeden z popełnianych przez nowicjuszy błędów i pierwszy przypadek w którym można zauważyć że zasadne jest posiadanie zarówno **null** jak i **undefined** w języku.

Poprawne jest testowanie za pomocą

```
if ( a[100] !== undefined )
```

lub

```
if ( typeof a[100] !== 'undefined' )
```

(przypominamy że operator **typeof** zwraca wartości literalne).

Na wykładzie omówimy również podstawowy interfejs komunikacyjny tablic, wprowadzając wcześniej [składnię typu lambda](#) dla funkcji (o funkcjach będzie mowa w kolejnej sekcji)

- [slice](#)
- [splice](#)
- [filter](#)
- [map](#)
- [reduce](#)
- [join](#)

Omówimy też enumerowanie tablic za pomocą konstrukcji [for-of](#) i [for-in](#). Na kolejnym wykładzie dowiemy się jak umożliwić enumerowanie zawartości dowolnego obiektu za pomocą for-of.

Wyjątki

Na wykładzie omówimy mechanizm [wyjątków](#).

```
try {  
    throw new Error('wyjątek');  
}  
catch ( e ) {  
    console.log( e.message );  
}
```

Dialekty zawężające

Oryginalne sformułowanie składni Javascript może w przypadkach brzegowych prowadzić do nieścisłości. Stąd w wersji 5 języka dodano możliwość opcjonalnego włączenia ograniczenia składni do tzw. [trybu ścisłego](#) (Strict mode). W trakcie wykładu zobaczymy przykłady.

Innym dialektem którego istnienie warto odnotować (aczkolwiek niekoniecznie posługiwać się nim na co dzień) jest [asm.js](#). Ten wzmiankowany już na wykładzie dialekt jest używany zwykle w sytuacjach gdy Javascript powstaje jako kompilat języka wyższego poziomu i ma na celu dodanie optymalizacji wydajnościowych. Asm.js jest domyślnie dostępny we współczesnych przeglądarkach.

JSON

[Javascript Object Notation](#) to interoperacyjny format wymiany danych o składni wywiedzionej z Javascript. Javascript posiada wsparcie dla JSON, na wykładzie zademonstrujemy funkcje **JSON.stringify** i **JSON.parse**.

Funkcje jako obiekty pierwszoklasowe

Jako język funkcyjny, Javascript traktuje funkcje jako obiekty pierwszoklasowe (ang. [first-class citizen](#)).

W trakcie wykładu obejrzymy podstawowe przykłady, a temat funkcji będziemy kontynuować na kolejnych zajęciach:

- sposoby definicji funkcji

```
var f = function() {  
  
}  
  
function g() {  
  
}  
  
var h = new Function();
```

-