Wybrane elementy praktyki projektowania oprogramowania
Wykład 03/15
JavaScript, podstawy języka (2)

Wiktor Zychla 2019/2020

Spis treści

Podstawy języka, ciąg dalszy …

String templates

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

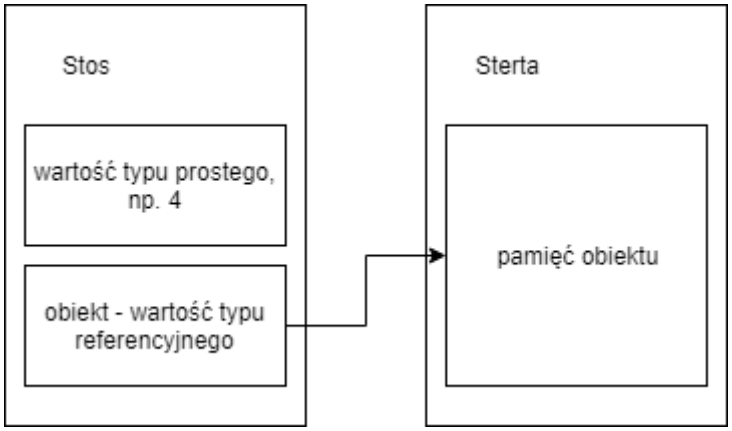
```
var n = 1, m = 2;
var string = `szablon z wartościami ${n}, ${m} i ${n+m}`;

console.log( string );
```

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 stercie (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 wnętrze 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 wywołaniania ${n}`);
```

getter/seter – 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 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() );
```

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 częściej popełnianych przez nowicjuszy błędów.

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

- slice
- splice
- filter
- map

- reduce

Omówimy też enumerowanie tablic za pomocą konstrukcji for-of i for-in.

Wyjątki

Na wykładzie omówimy mechanizm wyjątków.

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

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:

- przekazywanie funkcji do funkcji i zwracanie funkcji z funkcji
- odczytywanie argumentów wewnątrz funkcji
- zmienna liczba argumentów
- domknięcia (closures)
- lambda wyrażenia