Wybrane elementy praktyki projektowania oprogramowania

Wykład 05/15
JavaScript, obiektowość prototypowa

Wiktor Zychla 2019/2020

1 Spis treści

2 Obiektowość

Obiekt = stan pamięci + metody do operowania na tej pamięci

Paradygmat programowania obiektowego wychodzi naprzeciw takiej typowej strukturze kodu w języku nie obiektowym, w którym struktura danych ma metody dedykowane do jej przetwarzania. Na język obiektowy można patrzeć jak na język imperatywny, w którym zaproponowana jest pewna *konwencja* pisania kodu strukturalnego.

Dla wygody, języki obiektowe posiadają pewne elementy *opcjonalne*, które ułatwiają tworzenie kodu ale nie są wymagane. Ich istnieje jest jednak często mylnie utożsamiane z *obiektowością*:

- Klasy
- Dziedziczenie
- Konstruktory, operator **new**

Mówiąc inaczej, jeśli język pozwala jakoś pozyskać referencję do struktury danych, a następnie wywołać metodę która przyjmuje obiekt struktury jako swój argument, to już możemy mówić o języku obiektowym i stosować do takiego języka całą wiedzę o tzw. wzorcach projektowych.

Projektowanie obiektowe = określanie odpowiedzialności obiektów (klas) i ich relacji względem siebie. Wszystkie dobre praktyki, zasady, wzorce sprowadzają się do tego jak właściwie rozdzielić odpowiedzialność na zbiór obiektów (klas).

W Javascript są trzy równoważne modele implementowania struktur obiektów, wynikające z trzech różnych sposobów konstruowania nowych instancji obiektów

1. `{}`
2. `new`
3. `Object.create()`

Proszę zwrócić uwagę, że w językach takich jak C# czy Java jest tylko jeden sposób tworzenia nowych instancji obiektów – to operator **`new`**.

3 Obiektowość z { }, bez dziedziczenia

Pierwsza możliwość to symulowanie konstruktorów przez funkcje tworzące obiekty za pomocą składni literalnej. Wadą takiego podejścia jest niepotrzebne zużycie pamięci na wielokrotne kopie tych samych metody (da się to obejść).

```
function Person(name, surname) {
    return {
        name    : name,
        surname : surname,
        say     : function() {
            return `${this.name} ${this.surname}`;
        }
    }
}

var p = Person('jan', 'kowalski');

console.log( p.say() );
```

Proszę we własnym zakresie spróbować w tym podejściu zaimplementować dziedziczenie (czyli możliwość zdefiniowania "podklasy" w której w implementacji "konstruktora" i metod można odwołać się do implementacji konstruktora i metod z "klasy bazowej").

4 Obiektowość prototypowa

Obiektowość prototypowa (*prototypal inheritance*) (łańcuch prototypów, darmowy podręcznik: You don't know JS) pozwala rozwiązać problem "współdzielenia" kodu przez wiele instancji obiektów, zestawiając je w łańcuchy w których każdy obiekt wskazuje na swój prototyp.

Obiektowi można ustawić/zmienić prototyp w trakcie działania programu (**Object.setPrototypeOf**), można też odczytać prototyp istniejącego obiektu (**Object.getPrototypeOf**).

```
var p = {
    name : 'jan',
    say : function() {
        return this.name;
    }
};

var q = {}

// prototypem q będzie p
Object.setPrototypeOf( q, p );

q.name = 'tomasz';

// q ma już metodę say, bo pochodzi ona z prototypu
console.log( q.say() );
```

Prosty eksperyment z pomocą funkcji

```
function getLastProto(o) {
    var p = o;
    do {
        o = p;
        p = Object.getPrototypeOf(o);
    } while (p);

    return o;
}
```

pokazuje że wszystkie obiekty Javascript mają jedną, tę samą instancję obiektu jako swój prototyp (jest to ładna analogia do języków w których mamy hierarchie typów z jednym typem bazowym dla całej hierarchii - tu mamy jeden obiekt który jest wspólnym prototypem wszystkich obiektów, to w nim znajdują się wyjściowe implementacje m.in. (**toString**) - dlatego te "podziedziczone" metody są dostępne dla wszystkich obiektów).

Tym wspólnym prototypem wszystkich obiektów jest obiekt **Object.prototype**. Można dodawać do niego składowe (funkcje, właściwości), które są automatycznie „dziedziczone” w dół łańcucha prototypów.

Można w ten sposób rozszerzyć dowolny inny obiekt prototypowy, ograniczając zasięg dostępności składowych do tego fragmentu łańcucha:

```
String.prototype.reverse = function() {
    return this.split('').reverse().join("");
}

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

5 Obiektowość przez Object.create()

Funkcja **Object.create()** tworzy nową instancję obiektu i ustawia jej wskazany obiekt jako prototyp.

"Konstruktor" - musi być rozdzielony na tworzenie przez Object.create (poprawne zestawienie łańcucha prototypów) i metodę init która jedynie inicuje stan obiektu (można to rozwiązać inaczej ale to jest stosunkowo eleganckie rozwiązanie):

```
var person = {
    init : function(name, surname) {
        this.name    = name;
        this.surname = surname;
    },
    say  : function() {
        return `${this.name} ${this.surname}`;
    }
}

var p = Object.create( person );
p.init( 'jan', 'kowalski' );

console.log( p.say() );
```

"Dziedziczenie" jest jak najbardziej możliwe

```
var worker = Object.create( person );
worker.init = function( name, surname, age ) {
    // "wywołanie konstruktora klasy bazowej"
    person.init.call( this, name, surname );
    this.age = age;
}
worker.say = function() {
    // "wywołanie metody z klasy bazowej"
    var _ = person.say.call( this );
    return `${_} ${this.age}`;
}

var w = Object.create( worker );
w.init('tomasz','malinowski',48);
console.log( w.say() );
```

6 Obiektowość z new

Funkcja konstruktorowa

```
var Person = function(name, surname) {
    this.name = name;
    this.surname = surname;
}
Person.prototype.say = function() {
    return `${this.name} ${this.surname}`;
}

var p = new Person('jan', 'kowalski');
console.log( p.say() );
```

„Dziedziczenie”

```
var Worker = function(name, surname, age) {
    // wywołanie bazowej funkcji konstruktorowej
    Person.call( this, name, surname );
    this.age = age;
}

// powiązanie łańcucha prototypów
Worker.prototype = Object.create( Person.prototype );

Worker.prototype.say = function() {
    // "wywołanie metody z klasy bazowej"
    var _ = Person.prototype.say.call( this );
    return `${_} ${this.age}`;
}

var w = new Worker('jan', 'kowalski', 48);
console.log( w.say() );
```

7 Równoważność obu sposobów

Oba sposoby implementacji obiektowości, ten w którym **Object.create()** jest pierwotne i ten w którym **new** jest pierwotne, są równoważne (to znaczy że w języku mógłby istnieć tylko jeden z nich, bo drugi da się wyrazić za jego pomocą).

Wyrażenie **new** za pomocą **Object.create**

```
var Person = function(name, surname) {
    this.name = name;
    this.surname = surname;
}
Person.prototype.say = function() {
    return `${this.name} ${this.surname}`;
}

// alternatywa dla new f()
// wyrażona przy pomocy Object.create
function New( f, ...args ) {
    var _ = Object.create( f.prototype );
    var o = f.apply( _, args );
    if ( o )
        return o;
    else
        return _;
}

var p = New( Person, 'jan', 'kowalski' );
console.log( p.say() );
```

Wyrażenie **Object.create** za pomocą **new**

```
var person = {
    init : function(name, surname) {
        this.name    = name;
        this.surname = surname;
    },
    say  : function() {
        return `${this.name} ${this.surname}`;
    }
}

// alternatywa dla Object.create( p )
// wyrażona przy pomocy "new"
function ObjectCreate( p ) {
    var f = function() { };
    f.prototype = p;
    return new f();
```

```
}

var p = ObjectCreate( person );
p.init('jan', 'kowalski');
console.log( p.say() );
```

8 Lukier syntaktyczny definicji klas

W rozszerzeniu dialektu Javascript ES2016, otrzymaliśmy lukier syntaktyczny na funkcje konstruktorowe/**new** : class i extends:

```
class Person {
    constructor(name, surname) {
        this.name = name;
        this.surname = surname;
    }

    say() {
        return `${this.name} ${this.surname}`;
    }
}

class Worker extends Person {
    constructor(name, surname, age ) {
        super(name, surname);
        this.age = age;
    }

    say() {
        // "wywołanie metody z klasy bazowej"
        var _ = super.say();
        return `${_} ${this.age}`;
    }
}

var w = new Worker('tomasz', 'malinowski', 48);
console.log( w.say() );
```