Wybrane elementy praktyki projektowania oprogramowania

Wykład 04/15
JavaScript, obiektowość prototypowa

Wiktor Zychla 2023/2024

1 Spis treści

2 Paradygmat obiektowy

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:

```
struct Person {
   Name: string;
   DateOfBirth: date;
}

void PrintPerson( Person Person, string Format ) {
    ...
}

int Age( Person Person ) {
    ...
}
```

Taki strukturalny kod w typowym języku obiektowym, w którym struktura kodu oparta jest o tzw. **klasy**, wyrażałby się tak:

```
class Person {
    Name: string;
    DateOfBirth: date;

    void PrintPerson( string Format ) {
        ...
    }

    int Age() {
        ...
    }
}
```

Na język obiektowy można patrzeć jak na język imperatywny, w którym zaproponowana jest pewna *konwencja* pisania kodu strukturalnego.

Dla wygody, języki obiektowe posiadają pewne mechanizmy **opcjonalne**, które ułatwiają tworzenie kodu, ale nie są wymagane.

Ich istnienie bywa jednak **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 szczególności, powtórzmy to jeszcze raz – **dziedziczenie nie jest techniką która jest w języku obiektowym niezbędna.** Z **dziedziczenia** płyną natomiast pewne **korzyści**:

1. Metody zaimplementowane w klasie są współdzielone przez wszystkie wystąpienia (instancje) danej klasy, co pozwala na **oszczędzanie pamięci**. Często stosowaną techniką implementacyjną jest tzw. tablica funkcji wirtualnych. Alternatywą byłoby posiadanie przez każdy obiekt kopii każdej metody, ale prowadziłoby to do niepotrzebnego zużywania pamięci. Ale paradygmat obiektowy posługuje się pojęciem **delegowania**, które polega na tym że obiekt posiada referencję do innego obiektu, do którego metod może się odwoływać nawet w sytuacji kiedy sam takich metod nie posiada – i jest to jeden ze sposobów unikania tego nadmiernego zużywania pamięci
2. Jeżeli typ **Child** jest podtypem typu **Parent**, to każde wystąpienie obiektu **Child** jest równocześnie wystąpieniem obiektu typu **Parent**. W sytuacji gdy funkcja F spodziewa się parametru typu **Parent** lub zwraca wartość typu **Parent**, kompilator języka obiektowego, który kontroluje typy w trakcie kompilacji, pozwoli zamiast **Parent** przyjąć/zwrócić obiekt typu **Child**. Dziedziczenie jest więc jednym z dodatkowych elementów **kontroli typów podczas kompilacji**.

W Javascript druga z tych korzyści nie ma zastosowania, ponieważ nie ma „kontroli typów podczas kompilacji”. Jeżeli dwa niepowiązane ze sobą w żaden sposób obiekty mają „podobny wygląd” (pola/metody), to obu można użyć w tym samym kontekście i do tego nie jest potrzebna żadna kontrola typów:

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

var car = {
  brand: 'skoda',
  say: function() {
    return this.brand;
  }
}

function describe(item) {
  console.log( item.say() );
```

```
}

describe( person );
describe( car );
```

O takim luźnym podejściu do specyfikacji w którym zamiast szablonu/klasy, który określa jaką obiekt ma strukturę, jest prosta zasada – jeżeli obiekt ma jakąś metodę/pole to je ma, bez związku z tym czy ten obiekt ma jakiś „typ", mówimy Duck Typing („*if it walks like a duck and it quacks like a duck, then it must be a duck*").

Jeśli natomiast chodzi o pierwszą z korzyści z dziedziczenia, czyli oszczędzanie zużycia pamięci, to pokażemy jak **obiektowość prototypowa** proponuje inne rozwiązanie niż mechanizm oparty na klasach.

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 z trzech możliwych technik programowania obiektowego w JS, to *udawanie konstruktorów* przez proste funkcje tworzące obiekty przy pomocy składni literalnej. Wadą takiego podejścia jest niepotrzebne zużycie pamięci na wielokrotne kopie tych samych metod.

```
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"). Nie jest to trudne.

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.

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

// say pochodzi z prototypu, name z q
q.name = 'tomasz';
console.log( q.say() );

// say i name pochodzą z q
q.say = function() {
   return this.name + ' from q'
}
console.log( q.say() );

// say pochodzi z q, name z prototypu
delete q.name;
console.log( q.say() );
```

4.2 Łańcuch prototypów

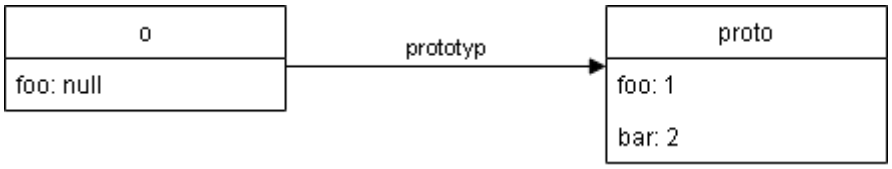
Zasada jest taka że przy odwołaniu do składowej obiektu, **o.foo** lub **o.foo()**, do ustalenia czym jest **foo** dla obiektu **o** stosowany jest następujący algorytm:

- Dla odczytu wartości powtarzaj:
    - Sprawdź czy obiekt o ma pole/metodę **foo**
    - Sprawdź czy prototyp o ma pole/metodę **foo**
    - Sprawdź czy prototyp prototypu ma pole/metodę **foo**
    - Itd. aż skończy się łańcuch prototypów
- Dla zapisu wartości:
    - Zapisz wartość **foo** bezpośrednio w obiekcie **o**

Co znaczy „sprawdź czy obiekt ma pole/metodę **foo**? Kluczowe jest odróżnienie **wartości** od **braku wartości**. To też główny powód, dla którego w języku mamy zarówno **null** jak i **undefined**:

- **null** oznacza że wartość jest w danym obiekcie (tylko jest pusta) i nie trzeba kontynuować poszukiwania w łańcuchu prototypów
- **undefined** oznacza że wartości nie ma w danym obiekcie i trzeba kontynuować przeszukiwanie łańcucha prototypów

```
var o = {
  foo: null
}

var proto = {
  foo: 1,
  bar: 2
}

Object.setPrototypeOf(o, proto);

// foo pochodzi z obiektu o
console.log( o.foo );
// bar pochodzi z obiektu proto,
// bo bar nie występuje w foo
console.log( o.bar );
```



4.3 Reużywanie prototypów

Ponieważ ten sam obiekt może być prototypem dla wielu obiektów, można już zauważyć efekt oszczędzania pamięci. Konwencja, której warto przestrzegać, byłaby taka:

- Każdy obiekt ma swój własny stan (pola)
- Obiekty o tym samym prototypie mogą współdzielić metody (bo ciała metod ma prototyp)

```
var personProto = {
  say: function() {
    return this.name;
  }
}

var p1 = {
  name: 'jan'
}
Object.setPrototypeOf( p1, personProto );

var p2 = {
  name: 'tomasz'
}
Object.setPrototypeOf( p2, personProto );

console.log( p1.say() );
console.log( p2.say() );
```

4.4 Zbieżność łańcucha prototypów

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.

Ta konwencja, w której prototyp jakiejś grupy obiektów jest wprost dostępny jako **XXX.prototype,** gdzie XXX jest literałem reprezentującym coś czego jeszcze nie nazwaliśmy, wyjaśni się w następnym rozdziale – XXX będzie **funkcją konstruktorową**, za pomocą której utworzono obiekt któremu przypisano ten prototyp.

4.5 Ograniczenia

Obiekt może mieć pusty prototyp. W szczególności, ten prototyp wszystkich obiektów, czyli Object.prototype, ma pusty prototyp. Dzięki temu łańcuch prototypów zawsze się kończy, więc przeszkuwanie łańcucha w poszukiwaniu składowej się kończy.

Łańcuch prototypów nie może mieć cykli. Gdyby łańcuch prototypów pozwalał na cykle, w trakcie działania kodu można by spodziewać się pętli nieskończonych na przeszukiwaniu cyklicznych łańcuchów prototypów. Na szczęście nad tym czuwa samo środowisko uruchomieniowe:

```
var o = {}
var p = {}

Object.setPrototypeOf( o, p );
Object.setPrototypeOf( p, o );


> Object.setPrototypeOf( p, o );
       ^
TypeError: Cyclic __proto__ value
    at Function.setPrototypeOf (<anonymous>)
```

5 Obiektowość z `new`

Funkcja konstruktorowa i **new:** każda funkcja w Javascript może być zawołana wprost oraz z **new**. Zawołanie funkcji **Foo** z **new** zmienia zachowanie środowiska uruchomieniowego:

1. Tworzony jest nowy pusty obiekt **{}**
2. Obiekt **Foo.prototype** jest przypisywany jako prototyp tego nowego obiektu
3. Funkcja **Foo** jest zawołana w taki sposób, że ten nowy pusty obiekt jest związany do **this** w środku ciała funkcji,
4. jeśli funkcja nie zwraca żadnej wartości, zwracany jest ten nowo zainicjowany obiekt
5. jeśli funkcja zwraca wartość obiektu typu referencyjnego, zwracany jest ten obiekt (z tej właściwości funkcji konstruktorowych korzysta się bardzo rzadko)

W przepisie tym jest również wskazówka gdzie szukać prototypu do takich nowo tworzonych obiektów: dla każdej funkcji konstruktorowej **X**, obiekt **X.prototype** staje się prototypem wszystkich obiektów tworzonych przez X.

To jest zwykły obiekt, można mu dodawać składowe a nawet go całkiem napisać, nawet w trakcie działania programu.

W ten sposób można uzyskać efekt zmniejszenia zużycia pamięci, natomiast – trzeba o to mimo wszystko zadbać. Proszę porównać to podejście w którym każdy obiekt ma własną kopię funkcji **say** …

```
function Foo() {
  this.prop = 1;
  this.say  = function() {
    return this.prop;
  }
}

var foo1 = new Foo();
var foo2 = new Foo();

console.log( foo1.say() );
```

… z tym podejściem gdzie trzymamy się zasady „stan (pola) w obiekcie, metody w prototypie”:

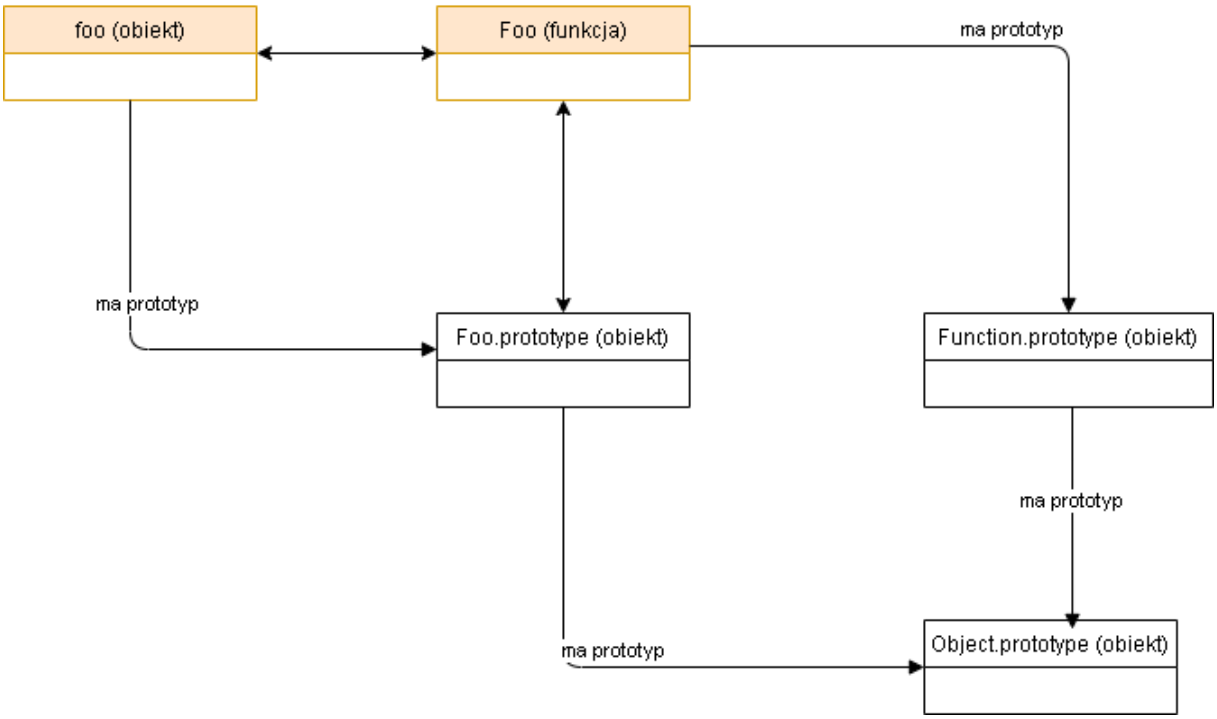
```
function Foo() {
  this.prop = 1;
}
Foo.prototype.say = function() {
  return this.prop;
}

var foo1 = new Foo();
var foo2 = new Foo();
```

```
console.log( foo1.say() );
```

Na poniższym diagramie zilustrowano zależności między obiektami **foo**, **Foo**, **Foo.prototype** oraz dodatkowo umieszczono inne obiekty, które są ich prototypami:

- obiekt **foo** i funkcja **Foo** mają taki związek że obiekt został „wyprodukowany” przez funkcję
- funkcja **Foo** i obiekt **Foo.prototype** mają taki związek że funkcja dodaje ten obiekt jako prototyp do wszystkich tworzonych przez siebie obiektów
- prototypem **foo** jest więc **Foo.prototype**
- a prototypy pozostałych?
    - **Foo** jako funkcja ma swój prototyp – **Function.prototype**
    - **Foo.prototype** jako obiekt ma swój prototyp – **Object.prototype**
    - **Function.prototype** jako obiekt ma swój prototyp – **Object.prototype**
    - **Object.prototype** ma pusty prototyp

var foo = new Foo()



W ten sposób można budować interfejs programistyczny przypominający klasy z języków obiektowych z klasami:

```
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() );
```

Sporo obiektów, z którymi pracujemy w żywym kodzie ma jako prototypy obiekty z właściwych funkcji konstruktorowych.

```
var o = {}
// true
console.log( Object.getPrototypeOf(o) === Object.prototype );

function foo() {}
// true
console.log( Object.getPrototypeOf(foo) === Function.prototype );

var s = 'ala ma kota';
// true
console.log( Object.getPrototypeOf(s) === String.prototype );
```

Zauważmy, że dodanie nowej funkcjonalności (pola lub funkcji) do prototypu automatycznie dodaje ją do **wszystkich** obiektów o tym prototypie, bez względu na to czy najpierw utworzy się te obiekty a potem rozszerzy prototyp czy najpierw utworzy prototyp a potem obiekty. To trochę tak, jakby w klasycznym języku obiektowym w trakcie działania programu móc do klasy dodać dynamicznie jakąś

metodę i wszystkie instancje tej klasy automatycznie dostałyby tę nową metodę (w klasycznych językach obiektowych tak się nie da)!

Zauważmy też, że można w ten sposób nawet rozszerzyć dowolny obiekt prototypowy z biblioteki języka, dodając wszystkim istniejącym obiektom o takim prototypie nową funkcjonalność:

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

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

W przykładzie rozszerzenie jest dodane do **String.prototype** ale oczywiście technicznie możliwe jest dodanie dowolnego rozszerzenia nawet do **Object.prototype** (aczkolwiek traktowane to jest jako praktyka kontrowersyjna).

Warto zwrócić uwagę na to że **każda funkcja** może być w takim razie wywołana z **new** lub **bez new,** nawet omyłkowo. Za pomocą sprawdzenia **new.target** można w ciele funkcji odróżnić oba przypadki

```
function foo() {
    if(new.target === undefined)
        console.log('zwykłe wywołanie');
    else
        console.log('wywołanie z new');
}

new foo();
foo();
```

6 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() );
```

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() );
```

W tej konwencji **class Person** jest lukrem na **function Person** (funkcji konstruktorowej), ze wszelkimi konsekwencjami, w szczególności istnieje **Person.prototype** i jest on przypisywany jako prototyp wszystkim obiektom tworzonym przez konstruktor **Person**.

Jednym z bardziej zaskakujących faktów dotyczących określania relacji dziedziczenia jest to że skoro ta relacja jest dynamiczna, to można na przykład dziedziczyć z wartości zwracanej przez funkcję (sic!):

```
function Singleton() {
    return class Singleton {
        static instance;

        static getInstance() {
            if (!this.instance) this.instance = new this();
```

```
            return this.instance;
        }
    }
}

// sic! dziedziczenie z wartości zwracanej z funkcji!
class Car extends Singleton() {
  paint(color){
    this._color = color;
  }
}

// inna klasa dziedziczy swoją "kopię"
class Person extends Singleton() {
  print() {
    // osoby nie mają koloru
    return this._color;
  }
}

// car1 i car2 to ten sam obiekt
let car1 = Car.getInstance();
car1.paint('red');

let car2 = Car.getInstance();
console.log( car2._color );

// person to już inny obiekt
let person = Person.getInstance();
console.log( person._color );
```

9 Przykład – drzewo binarne

W nawiązaniu do przykładu, którym zakończono poprzedni wykład, tak wyglądałoby drzewo binarne zaimplementowane przy użyciu funkcji konstruktorowej:

```
function Tree(val, left, right) {
  this.left  = left;
  this.right = right;
  this.val   = val;
}

Tree.prototype[Symbol.iterator] = function*() {
  yield this.val;
  if ( this.left )  yield* this.left;
  if ( this.right ) yield* this.right;
}

var root = new Tree( 1, new Tree( 2, new Tree( 3 ) ), new Tree( 4 ));

for ( var e of root ) {
  console.log( e );
}
```

a tak przy użyciu lukru syntaktycznego z **class**

```
class Tree {
  constructor (val, left, right) {
    this.left  = left;
    this.right = right;
    this.val   = val;
  }

  [Symbol.iterator] = function*() {
    yield this.val;
    if ( this.left )  yield* this.left;
    if ( this.right ) yield* this.right;
  }
}

var root = new Tree( 1, new Tree( 2, new Tree( 3 ) ), new Tree( 4 ));

for ( var e of root ) {
  console.log( e );
}
```