Wybrane elementy praktyki projektowania oprogramowania
Zestaw 4

Javascript - dziedziczenie prototypowe

2024-10-29

Liczba punktów do zdobycia: **8/35**
Zestaw ważny do: 2024-11-12

1. (**1p**) W trakcie wykładu pokazano funkcję **getLastProto**, za pomocą której można sprawdzić do jakiego obiektu zbiega łańcuch prototypów. Wykonać eksperyment dla kilku obiektów i potwierdzić (sprawdzając równość referencji) że istotnie, łańcuch prototypów zbiega do jednego i tego samego obiektu.

2. (**1p**) Jak odróżnić składową (pole/funkcję) która pochodzi z obiektu od takiej która pochodzi z prototypu?

```
var p = {
  name: 'jan'
}
var q = {
  surname: 'kowalski'
}

Object.setPrototypeOf( p, q );

console.log( p.name );
console.log( p.surname );
```

   Formalnie, napisać prostą funkcję która dla zadanego obiektu i dla nazwy składowej, zwróci wartość logiczną prawda/fałsz, w zależności od tego czy składowa pochodzi z bezpośrednio obiektu czy z prototypu. Napisać fragmenty kodu które wyenumerują pola/funkcje obiektu:

   - tylko pola/funkcje występujące w obiekcie
   - pola/funkcje występujące w obiekcie oraz jego łańcuchu prototypów

3. (**1p**) W przykładzie z wykładu dotyczącym funkcji konstruktorowych, definiuje się funkcję konstruktorową **Person** a następnie funkcję konstruktorową **Worker**, w taki sposób żeby uzyskać efekt "dziedziczenia" znany z języków opartych o klasy. Powiązanie łańcuchów prototypów obu funkcji odbywa się w następujący sposób:

```
var Person = ...
var Worker = ...

Worker.prototype = Object.create( Person.prototype );
```

Wytłumaczyć zasadność/prawidłowość takiego podejścia. W szczególności, dlaczego nie powinno się tego łańcucha prototypów zestawiać tak:

```
var Person = ...
var Worker = ...

Worker.prototype = Person.prototype;
```

ani tak

```
var Person = ...
var Worker = ...

Worker.prototype = new Person();
```

? Jakie byłyby wady obu tych sposobów, których nie ma to prawidłowe podejście?

4. (**1p**) Czy wartości typów prostych też mają prototypy? Czy to znaczy że one są obiektami i można im dodawać dowolne pola/funkcje, jak wszystkim obiektom? Wyjaśnić wynik działania ostatniej linii poniższego programu:

```
var n = 1;

// liczba ma prototyp?
console.log( typeof Object.getPrototypeOf( n ) );

// można jej dopisać pole/funkcję?
n.foo = 'foo';
console.log( n.foo );
```

5. (**1p**) Na dowolnym przykładzie zademonstrować jak można uzyskać efekt "składowych prywatnych" znany z wielu języków obiektowych.

   Formalnie: zdefiniować funkcję konstruktorową `Foo`, do jej prototypu dodać metodę publiczną `Bar`, którą można zawołać na nowo tworzonych instancjach obiektów, ale w ciele funkcji `Bar` zawołać funkcję `Qux` która jest funkcją prywatną (czyli że funkcji `Qux` nie da się zawołać ani wprost na instancjach `Foo` ani w żaden inny sposób niż tylko z wewnątrz metody publicznej `Bar`).

   Wskazówka: zapewne funkcję można jakoś ukryć w domknięciu. Przypomnieć sobie techniki programowania funkcyjnego z poprzednich wykładów.

   Nieprawidłowe podejście, takie w którym zarówno metody `Bar` jak i `Qux` są publiczne, zaprezentowano poniżej

```
function Foo() {

}

Foo.prototype.Bar = function() {
    this.Qux();
}

Foo.prototype.Qux = function() {
    console.log( "Foo::Qux");
```

```
}

var foo = new Foo();
foo.Bar();

// ale w tym, niepoprawnym rozwiązaniu można też
foo.Qux();
```

6. (**3p**) Węzeł drzewa binarnego (`Tree`) to obiekt który przechowuje referencje do swojego lewego i prawego syna oraz wartość (np. liczba lub napis). Proszę w notatkach do wykładu odnaleźć implementację takiego węzła i nauczyć się używać zdefiniowanej tam funkcji konstruktorowej.

   Na wykładzie pokazano jak do prototypu funkcji konstruktorowej dodać generator, który powoduje enumerowanie zawartości drzewa "w głąb":

```
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 );
}

// 1 2 3 4
```

   Państwa zadaniem jest zaproponować definicję iteratora (zaimplementowanego wprost jako funkcja zwracająca obiekt zawierający `next`... lub jako generator z `yield`), który enumeruje zawartość drzewa "wszerz". Dla podanego wyżej przykładowego drzewa wynikiem enumeracji powinno być `1 4 2 3` i oczywiście w ogólnym przypadku, wyniki enumerowania "w głąb" i "wszerz" są różne, poza tym że w obu przypadkach pierwszym zwróconym wynikiem może być wartość z korzenia drzewa.

   (Oczywiście kolejność enumeracji zależy nie tylko od tego czy enumeruje się "wszerz" czy "w głąb" ale też od tego czy odwiedza się najpierw lewe czy prawe poddrzewo oraz od tego czy najpierw zwraca się węzeł czy jego podwęzły, dlatego proszę się nie sugerować kolejnością podaną wyżej jako jedyną prawidłową)

   Wskazówka: iterator nie będzie rekursywny, tak jak ten "w głąb". Zamiast tego, iterator wykorzystuje pomocniczą strukturę danych - **kolejkę** (jak za pomocą tablicy i funkcji `splice` w Javascript uzyskać efekt kolejki, czyli dodawanie elementów na koniec kolejki, ściąganie elementów z początku kolejki?) - oraz następujący algorytm

   (a) do kolejki włoż korzeń drzewa

   (b) powtarzaj dopóki kolejka jest niepusta

      i. wyjmij węzeł z kolejki

      ii. do kolejki włóż lewy i prawy podwęzeł (jeśli istnieją)

iii. zwróć (`value` lub `yield`) wartość węzła

Pytanie dodatkowe, niepunktowane: co by się stało, gdyby zamiast kolejki użyć stosu (dodawanie elementów na koniec stosu, ściągane elementów z końca stosu)?

Wiktor Zychla