Metody programowania

Wprowadzenie do platformy Java

Dr inż. Andrzej Grosser

Częstochowa, 2013

Spis treści

1. Klasy i obiekty

Celem tego rozdziału jest przedstawienie podstaw programowania obiektowego w Javie. Pokazano w jaki sposób stworzyć własne klasy, obiekty, jak definiować pola składowe i metody. Do napisania rozdziału wykorzystano pozycje literaturowe [1] i [2].

1.1. Klasy

Klasa jest szablonem w oparciu o który powstają obiekty - determinuje zawartość obiektu i jego zachowanie. Składniowo definicja klasy w Javie prezentuje się w sposób następujący (proszę zauważyć, że nie ma po definicji klasy średnika, tak jak w C++):

```
class NazwaKlasy {
    cialo klasy
}
```

Jak widać, po słowie kluczowym `class` pojawia się nazwa klasy (reguły nazewnictwa klas są w Javie takie same jak dla nazw zmiennych, przy czym nazwa klasy zaczyna się najczęściej od wielkiej litery - wynika to z powszechnie przyjętej konwencji), następnie między nawiasami klamrowymi znajduje się ciało klasy. W ciele klasy mogą się pojawić deklaracje atrybutów i metod. Na przykład:

```
1 class Osoba {
2 //definicja metody
3     void wypisz(){
4         System.out.println(nazwisko);
5     }
6 //definicja atrybutu
7     String nazwisko;
8 }
```

1.2. Obiekty

Obiekty są instancjami klas, zawierają konkretne dane. Obiekty w Javie są tworzone za pośrednictwem słowa kluczowego `new`. Zmienna w Javie nie przechowuje bezpośrednio

obiektu, jedynie wskazuje na niego (są więc tak naprawdę odpowiednikiem wskaźnika z C++). Odpowiednikiem słowa kluczowego `null_ptr` z C++[1] jest w Javie `null`. Tą wartością są domyślnie inicjowane zmienne obiektowe.

Dostęp do składowych obiektów jest realizowany za pomocą operatora dostępu do składowej (kropki). Na przykład:

```
1 class X {
2     int x = 10;
3     void f() {}
4 }
5 //...
6 X x = new X();
7 System.out.println(x.x);
8 x.f();
9 //...
```

Odwołanie do atrybutu lub wywowałnie metody na rzecz obiektu pustego (wskazującego na null) powoduje wywołanie w czasie wykonania wyjątku w trakcie wykonania programu - java.lang.NullPointerException.

1.2.1. Poziomy dostępu do składowych

W Javie wyróżnia się cztery sposoby ograniczenia dostępu do składowych: publiczny (wyróżniony słowem kluczowym `public`), chroniony (wyróżniony słowem kluczowym `protected`), prywatny (wyróżniony słowem kluczowym `private`) i pakietowy (bez słowa kluczowego, poziom domyślny).

W przypadku metod i atrybutów publicznych dostęp do tego rodzaju składowych jest możliwy wszędzie, do chronionych w klasie i klasach po niej dziedziczących, prywatny tylko w klasie, natomiast pakietowy w klasie i wewnątrz pakietu, w jakim ona została zdefiniowana.

W Javie modyfikatory odnoszą się tylko do elementów, które stoją bezpośrednio za nimi. Nie modyfikują dostępu do zdefiniowanych dalej w pliku składowych (tak jak w C++). W kodzie poniżej, wprowadzenie słowa `public` przed deklaracją atrybutu x nie spowodowało, zmiany poziomu dostępu do atrybutu y - jest do niego domyślny poziom dostępu (pakietowy):

[1] Słowo kluczowe null_ptr pojawiło się w nowym standardzie języka. Zostało ono wprowadzone ze względu na problemy z użyciem stałych 0 i NULL.

```
class Przyklad {
//składowa publiczna
public int x;
//składowa pakietowa
int y;
//składowa chroniona
protected int y;
//składowe prywatne
private int z;
private void f() {}
}
```

Jeżeli należy stworzyć więcej niż jeden element o tym samym poziomie dostępu, należy każdy z nich poprzedzić stosownym modyfikatorem - tak jak dla prywatnych składowych z i f().

1.3. Atrybuty

Atrybuty (zwane też polami składowymi) opisują dane klasy (i obiektów). Mogą być zarówno typu prostego, jak i obiektami klas. Deklaracja atrybutu wygląda tak samo jak deklaracja zmiennej wewnątrz definicji funkcji. Atrybuty przyjmują jako wartości początkowe wartości domyślne dla poszczególnych typów (zero dla typów całkowitych i rzeczywistych, fałsz dla typu logicznego, null dla obiektów). Można też nadawać jawnie wartości początkowe atrybutów (inaczej niż w C++, gdzie było to możliwe jedynie dla stałych typu całkowitego). Na przykład:

```
class Przyklad {
    int y;
    int x = 10;
    double z;
    double z2 = 12.0;
}
```

Możliwe jest nawet tworzenie w ten sposób obiektów klas:

```
class X{}
class Y {
     X x = new X();
}
```

Atrybutom można również nadawać wartości wewnątrz tak zwanych bloków inicjalizacyjnych. W blokach możliwe jest wykonywanie instrukcji. Bloki inicjacyjne są wywoływane przed konstruktorami.

W poniższym kodzie wykorzystano blok inicjalizacyjny do utworzenia i zainicjowania tablicy liczb całkowitych:

```
1 class Z {
2     {
3         int[] tab = new int[10];
4         for(int i = 0; i < 10; ++i)
5             tab[i] = i * i;
6     }
7 }
```

1.3.1. Atrybuty stałe

Do utworzenia obiektów stałych wewnątrz klas służy słowo kluczowe `final`. Nadawać wartość początkową tego typu pola można wewnątrz definicji klasy, w konstruktorze i blokach inicjowania. Po nadaniu wartości tego typu pól nie można modyfikować. Dla przykładu:

```
1 class Foo {
2     final int x = 10;
3     final int y;
4     final int z;
5     {
6         z = 5;
7     }
8
9     Foo() {
10        y = 12;
11    }
12    void f() {
13    //Niepoprawne np.
14    //x = 12;
15    }
```

Znaczenie tego słowa może być jednak mylące dla obiektów klas, których stan można modyfikować. Słowo final w tym przypadku oznacza, że referencja zaszyta wewnątrz nie będzie zmieniała się po konstrukcji obiektu, natomiast obiekt, na który pokazuje może

zmieniać swój stan. W istocie więc, są w tym przypadku pola tego rodzaju odpowiednikami stałych wskaźników z języka C++.

Dla przykładu w podanym kodzie można modyfikować pola obiektu na który wskazuje x, natomiast pola y i str (obiektów String po utworzeniu nie da się modyfikować) są naprawdę stałe:

```
1  class X {
2      int x = 10;
3  }
4
5  class Bar {
6    final X x = new X();
7    final String str;
8    final int y = 1;
9    void f() {
10       // Ok
11       x.x = 12;
12       //Niepoprawne
13       // x = new X();
14   }
15 }
```

1.4. Metody

Metody definiują operacje jakie można wykonać na obiekcie klasy. Składnia metod w Javie prezenutje się w sposób następujący:

```
typWartosciZwracanej nazwaMetody(listaArgumentowFormalnych) {
    cialo metody
}
```

Metody w Javie definiowane są zawsze wewnątrz definicji klasy. Nie oznacza to jednak, że są automatycznie traktowane jako metody rozwijane w miejscu wywołania (inline), to kompilator analizując kod potrafi wyodrębnić miejsca, w których metody mogą być rozwijane.

Metoda niestatyczna ma dostęp do składowych obiektu na rzecz, którego jest wywoływana. Ten obiekt jest przekazywany jako dodatkowy niejawny parametr wywołania metody. Można się do niego odnieść, podobnie jak w C++, za pomocą słowa kluczowego `this`:

```
class C {
    private int z;
    public void f(int p)
    {
        z = p;
        //lub równoważne
        this.z = p;
    }
}
```

Argumenty są przekazywane do metod przez wartość. Należy jednak pamiętać, że w przypadku obiektów klas, są przekazywane jedynie odniesienia do właściwych obiektów, oznacza to, że wewnątrz metod jest możliwa modyfikacja stanu obiektów klas. Parametry funkcji składowych mogą przyjmować wartości domyślne.

Metody mogą zawierać ciąg instrukcji. Dozowolone jest tworzenie zmiennych lokalnych, ich nazwy mogą przesłaniać nazwy pól składowych klasy (nie jest to jednak zalecana praktyka, gdyż może prowadzić do błędów). Na przykład:

```
class X {
  int x;
  void f() {
      // x wewnątrz f() przesłania pole x
      // wszelkie bezpośrednie użycie x będzie odnosiło się
      // do tej zmiennej.
      int x = 12;
      //Można jednak
      this.x = 12;
  }
}
```

Wartości zwracane są z metod za pomocą instrukcji powrotu `return`. W tym momencie należy zwrócić uwagę na zwrot z metody obiektów klas, problem polega w tym przypadku na tym, że w tym momencie narusza się tak naprawdę enkapsulację obiektu, gdyż przypisując tak zwróconą referencję do zmiennej, operuje się tak naprawdę na obiekcie prywatnym. Można to wytłumaczyć dokładniej na następującym fragmencie kodu:

```
class X {
    private int m_;
    public void setM(int m) {
        m_ = m;
    }
```

```
}

class Y {
    private X x = new X();
    X getX() {
        return x;
    }
}

//...
Y y = new Y();
X x1 = y.getX();
x1.setM(10);
//...
```

Źródłem kłopotów w tym przypadku jest linijka `x1.setM(10);`, w tym momencie jest zmieniana zawartość obiektu prywatnego obiektu y, bo na niego wskazuje zmienna x1. Rozwiązaniem jest tworzenie kopii obiektu. Definicja klasy Y powinna wyglądać następująco:

```
class X { int a; }

class Y {
    private X x = new X();
    X getX() {
        X cp = new X();
        cp.a = x.a;
        return cp;
    }
}
```

Przedstawione powyżej rozwiązanie nie jest jedynym, można też przeciążyć metodę clone(), wymagałoby to jednak implementacji interfejsu Cloneable.

1.4.1. Przeładowanie nazwy metody

W Javie, podobnie jak w C++, dozwolone jest przeładowanie nazwy metody - innymi słowy kilka metod ma tą samą nazwę, rozróżniane są na podstawie liczby i typów przekazywanych argumentów (ale nie wartości zwracanej). Ta właściwość jest szczególnie przydatna dla konstruktorów klas.

1.4.2. Konstruktory

Konstruktor jest metodą wywoływaną bezpośrednio po utworzeniu obiektu. Najczęściej służy do inicjowania danych obiektu. Nosi nazwę taką samą jak nazwa klasy.

```
class X {
    private double x;
    public X() {
        x = 11.0;
    }
}
```

W konstruktorach dozwolone jest wywołanie innego konstruktora, wykonuje się to za pośrednictwem słowa kluczowego `this`. Wywołanie wygląda wtedy jak

`this(lista parametrow aktualnych);`

Przy czym ważne jest też, żeby ta instrukcja pojawiła się jako pierwsza w ciele konstruktora, w którym powinno się wywołać inny konstruktor.

```
class X {
    private int a;
    public X(int m) {
        a = m;
    }
    public X() {
        this(0);
    }
}
```

1.5. Atrybuty i metody statyczne

Podobnie jak w C++ dozwolone jest w Javie tworzenie atrybutów i metod klasowych, dostępnych dla wszystkich instancji klasy (korzystanie z nich jest, zresztą możliwe nawet bez definicji jakiegokolwiek obiektu klasy, w której zostały zdefiniowane). Tego rodzaju atrybuty i metody są nazywane atrybutami i metodami statycznymi. Elementy statyczne poprzedza się słowem kluczowym `static`, może pojawić się przed specyfikatorem dostępu lub za:

```
class X {
  public static void m() {
  }
  static private int z;
}
```

Odwołanie to metod i składowych statycznych jest realizowane najczęściej za pomocą nazwy klasy, kropki i nazwy odpowiedniej składowej. Dozowolone jest także odwołanie za pomocą obiektu, tak jak dla zwykłych pól.

```
class X {
  static void f() {}
}
//...
X obj = new X();
X.f();
obj.f();
//...
```

W przypadku metod statycznych, ponieważ nie jest przekazywana referencja do obiektu `this`, nie ma możliwości odwołania się do składowych instancji obiektu. Można jedynie jawnie przekazywać obiekty. Na przykład nie można użyć następującej konstrukcji:

```
//To nie jest poprawny kod Javy
class X {
    private double m;
    public static double g() {
        //Błąd
        return this.m;
        //Również niepoprawne
        return m;
    }
}
```

Natomiast jak najbardziej poprawne jest (odwołanie x1.m i x2.m jest jak w porządku, gdyż klasy mają dostęp do składowych prywatnych obiektów swojej klasy - reguła taka sama, jak w C++):

```
class X {
  private double m;
  public static double f(X x1, X x2) {
      return x1.m + x2.m;
  }
```

1.6. Pakiety

Java umożliwia łączenie grup powiązanych ze sobą logicznie klas w pakiety. Pozwala to przede wszystkim na uniknięcie konfliktów nazw. Są, więc w pewnym sensie, odpowiednikiem przestrzeni nazw języka C++.

Klasy biblioteki standardowej Javy są zorganizowane w pakiety, których nazwa zaczyna się słowem `java`. Dla przykładu `java.lang` zawiera najbardziej podstawowe klasy takie jak `String`, `java.io` - klasy obsługujące wejście-wyjście, `java.utils` - klasy pomocnicze itd. Rozszerzenia do platformy Javy, które zostały ustandaryzowane zawzyczaj należą do pakietów, których nazwa zaczyna się od `javax`, na przykład `javax.swing`.

Do utworzenia pakietu służy słowo kluczowe `package`, za którym zapisuje się nazwę pakietu, całość kończy się średnikiem. Deklaracja ta musi pojawić jako pierwsza instrukcja w pliku[2]. Na przykład dla następującego kodu wszystkie składowe pliku będą częścią pakietu `com.icis.utils`:

```
1 package com.icis.utils;
2
3 class X{}
```

1.6.1. Wykorzystywanie pakietów wewnątrz programów

Poszczególne składowe pakietu powinny trafić do odpowiedniej struktury katalogowej. Kompilator nie sprawdza co prawda, czy elementy pakietów są umieszczone we właściwym katalogu, da radę skompilować taki kod. Maszyna wirtualna jednak nie będzie w stanie go uruchomić.

1.6.2. Dołączanie klas z pakietów

Klasa może używać wszystkich klas ze swojego pakietu, ale jedynie publicznych z innych pakietów. Dostęp do tych zewnętrznych klas jest możliwy na dwa sposoby:

- Pełne odwołanie z całą nazwą pakietu np:

```
1 com.icis.utils.X x = new com.icis.utils.X();
```

- Posłużenie się importem danej klasy[3]

```
1 import com.icis.utils.X;
2 //Teraz można po prostu:
3 X x = new X();
```

Możliwe jest też zaimportowanie składowych statycznych za pomocą statycznego importu:

[2] Można ją pominąć, spowoduje to utworzenie tak zwanego pakietu domyślnego.

[3] Możliwy jest import wszystkich publicznych elementów pakietu za pomocą: import com.icis.utils.*;

```
1 import static java.lang.System.*;
```

Nie trzeba się wtedy posługiwać nazwą klasy, żeby odwołać się do tych składowych np:

```
1 //Zamiast pisać
2 System.out.println("Abc");
3 //można podać po prostu:
4 out.println("Abc");
```

Literatura

[1] D. Flanagan. *Java in Nutshell*. O'Reilly Media, 2005.

[2] C. S. Horstmann and G. Cornell. *Core Java Volume I–Fundamentals*. Prentice Hall, 2011.