Metody programowania

Wprowadzenie do platformy Java

Dr inż. Andrzej Grosser

Częstochowa, 2013

Spis treści

1. Dziedziczenie

Prezentowany rozdział stanowi wprowadzenie do dziedziczenia i polimorfizmu w Javie. Opisano w nim także interfejsy, klasy wewnętrzne i abstrakcyjne. Do napisania materiałów wykorzystano książkę [1].

1.1. Dziedziczenie

Wiele razy istnieje potrzeba rozszerzania klas o dodatkowe możliwości i dane. Jednym ze sposobów realizacji tego celu jest dziedziczenie.

W Javie dziedziczenie realizowane jest przy użyciu słowa kluczowego `extends` - w nagłówku klasy po jej nazwie występuje to słowo kluczowa a następnie nazwa klasy, po której będzie wykonywane dziedziczenie. Na przykład, w kodzie zamieszczonym poniżej, Pracownik jest klasą nadrzędną dla Kierownik - klasa Kierownik dziedziczy wszystkie elementy klasy Pracownik:

```
1 class Pracownik {
2 protected String imie,nazwisko;
3 }
4 class Kierownik extends Pracownik{
5 private int szczebel;
6 }
```

W Javie dziedziczenie jest odpowiednikiem dziedziczenia publicznego (dziedziczenia interfejsu) z języka C++, w Javie nie da się zapisać dziedziczenia prywatnego i chronionego (dziedziczenia implementacji). Dodatkowym ograniczeniem (nieszczególnie dotkliwym, gdyż są interfejsy) jest brak możliwości dziedziczenia wielokrotnego - można podać tylko jedną klasę bazową.

W klasie pochodnej można bezpośrednio odwoływać się do składowych publicznych i chronionych klasy nadrzędnej. Natomiast do składowych prywatnych, jeżeli w ogóle jest możliwe, to tylko za pośrednictwem odpowiednich metod publicznych lub chronionych.

Dozwolone jest przesłanianie pól i metod z klas nadrzędnych (niezależnie od poziomu dostępu do składowej). Można też zmieniać poziom dostępu do przesłanianej składowej. Na przykład, poprawny kod będzie wyglądał następująco:

```
1 class X {
2   int o;
3   private int x;
4   protected int y;
5   public int z;
6 }
7 class Y extends X {
8   protected int o;
9   int x;
10  public int y;
11  private int z;
12 }
```

Można zabronić dziedziczenia po danej klasie oznaczając ją jako klasę finalną. Umieszcza się bezpośrednio przed definicją klasy słowo kluczowe `final`. Na przykład w kodzie poniżej klasa X jest klasą finalną, nie da się po niej dziedziczyć:

```
1 final class X {}
2 //Zabronione np:
3 //class Y extends X {}
```

1.1.1. Klasa Object

Domyślnie w Javie (w przeciwieństwie do C++) klasy dziedziczą po klasie Object (a dokładniej java.lang.Object). Object definiuje wiele ogólnych metod włączając metodę pozwalającą przedstawić obiekt w postaci łańcucha znaków (`toString`), metodę pozwalającą porównywać między sobą obiekty (`equals`[1]) czy metodę umożliwiającą uzyskanie liczby umożliwiającej przechowywanie obiektu w zbiorze (`hashCode`).

Klasa Object daje szereg metod, które można wykorzystać przy odpytywaniu o właściwości klasy podczas wykonania programu. Tego rodzaju technika programowania jest nazywana refleksją.

[1]Operator równości porównuje dla typów obiektowych jedynie referencje obiektów. Prawdziwe porównanie zawartości obiektów można robi się za pomocą equals

1.1.2. Wywoływanie metod z klasy nadrzędnej

Do wywoływania metod z klasy nadrzędnej należy posłużyć się słowem kluczowym `super`. Używa się tego słowa, w tym kontekście, jako odwołania do podobiektu klasy nadrzędnej. Na przykład:

```
1  class X {
2    public void f() {
3      System.out.println("Metoda X");
4    }
5  }
6  class Y extends X {
7     public void f() {
8        super.f();
9        System.out.println("Metoda Y");
10   }
11
12 }
```

Wywołanie `super.f()` spowoduje wywołanie tej metody z klasy nadrzędnej.

W przypadku wywołania konstruktorów, słowo `super` staje się synonimem dla nazwy konstruktora klasy bazowej (C++ używa się do tego celu listy inicjacyjnej). Należy też pamiętać, że ich wywołanie w konstruktorze klasy podrzędnej musi być pierwszą, wykonywaną w nich instrukcją, np.:

```
1  class X {
2     int m;
3     public X(int a) {
4        m = a;
5     }
6  }
7  class Y extends X {
8     int z;
9     public Y(int a, int b) {
10       super(a);
11       z = b;
12    }
13 }
```

1.2. Polimorfizm

1.2.1. Zastępowalność klasa nadrzędna - podrzędna

W przypadku dziedziczenia można mówić o relacji ”jest rodzajem” - obiekt klasy pochodnej może być także traktowany jako obiekt klasy nadrzędnej (relacja odwrotna - obiekt klasy nadrzędnej może być traktowany jako obiekt klasy pochodnej - nie zawsze jest prawdziwa). Na przykład:

```
1 class Pracownik{}
2 class Robotnik extends Pracownik {}
3 class Kierownik extends Pracownik {}
4 //...
5 Pracownik p = new Pracownik();
6 Pracownik k = new Kierownik();
7 Kierownik k2 = (Kierownik) k;
```

W przytoczonym w kodzie powyżej przypadku, kierownik jest zawsze rodzajem pracownika, natomiast nie każdy pracownik jest kierownikiem. Dlatego też można zawsze odwołać się do obiektu klasy pochodnej za pośrednictwem odniesienia do obiektu klasy nadrzędnej, natomiast odwołanie do obiektu klasy pochodnej za pośrednictwem obiektu klasy nadrzędnej wymaga rzutowania, gdyż nie zawsze jest bezpieczne (obiekt klasy pochodnej zawiera składowe nadrzędnej, natomiast obiekt nadrzędnej nie musi zawierać składowych klasy pochodnej). Rzutowanie jest wymagana nawet w sytuacji, gdy na pewno wiadomo, że jest wskazywany obiekt odpowiedniej klasy (tak jak dla k w instrukcji inicjującej k2).

Opisywana we wcześniejszym akapicie właściwość zastępowania obiektu klasy pochodnej obiektem klasy nadrzędnej, jest jedną z kluczowych dla zrozumienia polimorfizmu.

1.2.2. Operator instanceof

Operator `instanceof` służy do testowania w czasie wykonania programu, czy obiekt należy do określonego typu. Składnia jest następująca:

```
obiekt instanceof Typ
```

Wyrażenie to zwraca prawdę, gdy obiekt jest instancją (lub da się sprowadzić) do zapisanego po prawej stronie typu.

Dla przykładu w poniższym kodzie wyrażenie zawierające instanceof zwróci prawdę w przypadku dwóch pierwszych użyć (jest bezpośrednio typu X lub dziedziczy po nim), natomiast fałsz w przypadku ostatniego:

```
class X {
}
class Y extends X {
}
//...
X x = new X();
Y y = new Y();
if (x instanceof X) {
  System.out.println("Instancja klasy X");
}
//true, bo Y jest podklasą X
if (y instanceof X) {
  System.out.println("Instancja klasy X");
}
if (x instanceof Y) {
  System.out.println("Instancja klasy Y");
}
```

Najczęściej operator instanceof służy on do sprawdzenia czy konwersja rzutująca w dół hierarchii dziedziczenia będzie bezpieczna:

```
class X {}
class Y extends X {}
class Z extends X {}
//...
X obj1 = new Y();
X obj2 = new Z();
if (obj1 instanceof Y) {
    Y y = (Y) obj1;
    //...
}
if (obj2 instanceof Y) {
    Y y = (Y) obj2;
    //...
}
```

W powyższym kodzie rzutowanie w dół jest bezpieczne jedynie dla obiektu obj1, w przypadku obiektu obj2 próba rzutowania zakończyłaby się wyjątkiem.

1.2.3. Wiązanie dynamiczne

Ponieważ korzystając z odniesienia do obiektu klasy nadrzędnej można zawsze odnieść się do dowolnych obiektów klas pochodnych, pojawia się zagadnienie, która metoda będzie

w danym przypadku wywoływana, jeżeli w obydwu klasach jest zdefiniowana metoda o tej samej nazwie i tym samych typach pobieranych argumentów. Na przykład:

```
class X {
    public void f() {
        System.out.println("Metoda klasy X");
    }
}
class Y extends X{
    public void f() {
        System.out.println("Metoda klasy Y");
    }
}
//...
X x = new Y();
x.f();
```

W podanym powyżej kodzie obiekt X wskazuje na obiekt klasy pochodnej. Są, więc dla wywołania x.f(), dwie opcje - albo jest wywoływana metoda z klasy nadrzędnej, albo metoda z klasy pochodnej. Java wybiera tą drugą możliwość, będzie wywołana metoda z klasy pochodnej (gdyż obiekt x pokazuje tak naprawdę obiekt klasy pochodnej).

Wywołanie odpowiedniej metody jest rozstrzygane w czasie działania programu. Dopiero wtedy najczęściej można sprawdzić na co rzeczywiście wskazuje odniesienie do obiektu klasy nadrzędnej. Ta technika jest nazywana wiązaniem dynamicznym.

Częstym błędem w Javie jest przypadkowe przeładowanie metody zamiast przesłonięcia. Dlatego wprowadzono w Javie 1.5 adnotację[2] `@Override` pozwalającą kompilatorowi na wyśledzenie tego problemu. Na przykład:

```
class X {
    public void f(int a) {
        System.out.println("Metoda klasy X");
    }
}
class Y extends X{
    @Override
    public void f(int a) {
        System.out.println("Metoda klasy Y");
    }
}
```

[2]Adnotacje to rodzaj znaczników lub metadanych dodawanych do kodu źródłowego, które mogą być czytane przez kompilator

W sytuacji, gdyby metoda f() z klasy Y nie posiadała dokładnie tych samych parametrów jak metoda z klasy nadrzędnej, kod nie skompilowałby się i kompilator wyrzuciłby stosowny komunikat.

1.2.4. Metody finalne

W Javie wszystkie metody domyślnie są polimorficzne. Można jednak, podobnie jak w przypadku klas, zabronić modyfikacji metod, stosując słowo kluczowe `final` - żadna podklasa nie będzie miała możliwości przesłonienia takiej metody (oczywiście oprócz nazwy muszą się zgadzać i typy parametrów):

```
1 class X {
2     final void f() {}
3 }
4 class Y extends X {
5     //Źle:
6     //void f() {}
7     //Poprawne:
8     void f(int a) {}
9 }
```

Wywołanie metody finalnej jest efektywniejsze - kompilator może rozstrzygnąć wywołanie takiej metody już na etapie kompilacji - pozwala to także rozwijać tego rodzaju funkcje w miejscu wywołania.

1.3. Klasy abstrakcyjne

Klasy, które zawierają definicję przynajmniej jednej metody abstrakcyjnej, są nazywane klasami abstrakcyjnymi. W przeciwieństwie do C++, należy powiadomić kompilator o tym, że definiuje się klasę abstrakcyjną dodając przed definicję klasy słowo kluczowe `abstract`. Metody czysto abstrakcyjne deklarowane wewnątrz takiej klasy również oznacza się w ten sposób. Kompilator uzyskuje w ten sposób informację, że obiektów tej klasy nie można tworzyć - klasa jest utworzona jedynie w celu utworzenia interfejsu dla klas pochodnych. Na przykład:

```
1 abstract class Figura {
2     public abstract double pole();
3 }
4 class Kwadrat extends Figura {
```

```
5     public a;
6     public double pole() {
7         return a * a;
8     }
9 }
10 //...
11 // Ok:
12 Figura f = new Kwadrat();
13 double w = f.pole();
14 //Źle:
15 //Figura f = new Figura();
```

W podanym powyżej przykładzie klasa Figura zapewnia abstrakcyjny interfejs umożliwiający polimorficzne wywołanie metody pole(). Operacja ta dla abstrakcyjnej figury nie ma sensu, dlatego została poprzedzona słowem kluczowym abstract, klasa Figura zawiera jedynie deklarację takiej metody (podobnie jak w C++[3]), jej definicja będzie musiała się znaleźć w klasach pochodnych.

1.4. Interfejsy

Realizacja większości celów, jakie dałyby się zrealizować za pośrednictwem dziedziczenia wielokrotnego daję się w Javie zrealizować za pomocą interfejsów. Interfejsy są podobne do klas, ale jeśli chodzi o metody można wskazać następujące różnice:

- Wszystkie metody interfejsu są abstrakcyjne.
- Wszystkie metody zadeklarowane w interfejsie są automatycznie publiczne.
- Nie można definiować metod statycznych w interfejsie.

W Javie klasy mogą dziedziczyć jedynie po jednej klasie, natomiast mogą implementować wiele interfejsów:

```
1 class X {}
2 class Y extends X implements Runnable, Comparable {
3 /* Implementacje metod z interfejsów */
4 }
```

Interfejsy można traktować jak zbiór wymagań, które muszą spełnić klasy implementujące dany interfejs. Dla przykładu podany poniżej interfejs Runnable wymusza na klasach implementujących ten interfejs definicję metody run().

[3]Oczywiście klasy abstrakcyjne są realizowane w C++ za pośrednictwem metod czysto wirtualnych.

```
public interface Runnable {
  public void run();
}
```

Interfejsy mogą zawierać dane, ale są one traktowane jako publiczne, finalne i statyczne. Na przykład:

```
interface Przyklad {
  int x = 1;
  int y = 2;
}
class Foo implements Przyklad {
}
//...
int z = Foo.x;
```

1.5. Klasy wewnętrzne

Podobnie jak w C++, w Javie można zagnieżdżać klasy wewnątrz klasy. W ten sposób podkreśla się ścisły związek pomiędzy klasami (a nie obiektami - obiekt klasy, w której zagnieżdżono klasę wewnętrzną nie posiada podobiektu klasy zagnieżdżonej).

Można wśród klas wewnętrznych wyróżnić następujące przypadki:

1. Zagnieżdżenie wewnątrz definicji klasy. Obiekt klasy wewnętrznej w tym przypadku ma wgląd na obiekt klasy zewnętrznej, dlatego musi być tworzony za jego pośrednictwem:

   ```
   class X {
       int a;

       public class Y {
           int f() {
               return a;
           }
       }
   }
   //...
   X x = new X();
   X.Y y = x.new Y();
   ```

   Obiekt klasy Y posiada ukrytą referencję do obiektu klasy X i dlatego musi być tworzony za pośrednictwem obiektu klasy X.

2. Lokalne klasy (wewnątrz definicji metody), na przykład:

```
class X {
  public void f() {
     class Y {}
  }
}
```

Lokalne klasy wewnętrzne są użyteczne, gdy z danych definicji korzysta się jedynie w jednym miejscu.

3. Anonimowe klasy wewnętrzne, na przykład:

```
class X {
  public void f() {
       Runnable r = new Runnable() {
          public void run() {
             System.out.println("Uruchomione");
          }
       };
  }
}
```

W tym przypadku, nie jest oczywiście tworzony obiekt interfejsu Runnable, tworzony jest obiekt klasy anonimowej implementującej ten interfejs.

Anonimowe klasy są użyteczne w sytuacji, gdy konieczne jest utworzenie tylko jednego obiektu klasy. Klasy anonimowe nie mogą zawierać konstruktorów. Ewentualne parametry przekazywane są do konstruktora klasy nadrzędnej, na przykład:

```
abstract class X {
   int a,b;
   public X(int p1, int p2) {
      a = p1;
      b = p2;
   }
   abstract void f();
}
//...
X x = new X(1,2) {
  void f() {}
};
```

4. Statyczne klasy wewnętrzne, na przykład:

```
class X {
  public static class Y {
```

```
3     }
4 //...
5 X.Y y = new X.Y();
6 }
```

W tym przypadku klasa zagnieżdżona nie przechowuje odniesienia do obiektu klasy zewnętrznej, dlatego jest tworzona niezależnie (zagnieżdżenie służy jedynie do podkreślenia, że klasa zagnieżdżona jest klasą pomocniczą).

Literatura

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