Dziedziczenie w Javie

Janusz Jabłonowski

12 kwietnia 2011

Wprowadzenie

- Klasy modelują pojęcia z dziedziny obliczeń.
- Pojęcia bywają ze sobą powiązane - chcemy to wyrażać w programach.
- Można to robić nieskutecznie lub dobrze:
  - Komentarze,
  - Dziedziczenie.
- Relacja uszczegóławiania (w drugą stronę: uogólniania).

Znaczenie dziedziczenia

- Dotyczy znaczenia klas a nie ich implementacji.
- Implementacja ma być ukryta.
- Zmiany implementacji nie mogą powodować zmian zależności dziedziczenia.
- Negatywne przykłady:
  - Klasy Punkt i Ułamek,
  - Klasy Lista i Stos.

Znaczenie dziedziczenia cd

- Jedna z podstawowych własności podejścia obiektowego.
- Pozwala tworzyć hierarchie.
- Formy hierarchii.
  - drzewo lub las.
  - DAG lub las DAGóW.
- Realizacja:
  - podklasa dziedziczy wszystkie atrybuty i metody po nadklasie,
  - także te, które nadklasa sama odziedziczyła,
  - konstruktory się nie dziedziczą (ale to bardziej złożona historia).
- Terminologia:
  - nadklasa i podklasa,
  - klasa bazowa i klasa pochodna.

Relacje *is-a* i *has-a*

- Dziedziczenie odzwierciedla relację *is-a*, czyli bycia czymś.
- Każdy obiekt podklasy jest także obiektem nadklasy.
- To prosty test na poprawność relacji dziedziczenia.
- Przykład.
  - Nadklasa *Owoc*.
  - Podklasy *Jabłko* i *Gruszka*.
- Częsty błąd: mylenie z relacją *has-a*.
- Tragicznie zły przykład: *Koło* dziedziczące po *Samochodzie*.

Znaczenie dziedziczenia - c.d.

- Połączenie dwu sprzeczności.
- Chcemy żeby tworzone oprogramowanie było:
  - Zamknięte (żadnych zmian po zakończeniu testów).
  - Otwarte (używane programy muszą być modyfikowane).

Zasada podstawialności

- Zawsze można podstawić zamiast obiektu nadklasy obiekt podklasy.
- Barbara Liskov, “Data Abstraction and Hierarchy,” SIGPLAN Notices, 23,5, May, 1988.
- Przykład: tam gdzie oczekujemy *Owocu* zawsze można podstawić *Jabłko*.
- Ale jak to zaimplementować, skoro obiekt podklasy jest zwykle większy od obiektu nadklasy?

Typowe zastosowania dziedziczenia

- Specjalizowanie pojęć (*Prostokąt* jako specjalizacja *Czworokąta*).
- Specyfikowanie pożądanego interfejsu (klasy abstrakcyjne, interfejsy).
- Rozszerzanie funkcjonalności (*KoloroweOkienko* rozszerzające możliwości *CzarnoBiałegoOkienka*).
- Nigdy nie używamy do ograniczania funkcjonalności.

Podmienianie metod

- Podklasy mogą różnie realizować zobowiązania z nadklas.
- Przykład: metoda *śpiewaj* w klasach:
  - nadklasa *Ptak*,
  - podklasy *Wrona*,*Słowik*.
- Można podmieniać (przedefiniowywać, ang. override) metody w podklasach.
- To niezwykle ważny mechanizm - zapewnia *polimorfizm*.

Wielodziedziczenie

- Jednoczesne dziedziczenie po kilku klasach.
- Np. C++, w Javie właściwie nie ma.
- Bardzo prosta i przekonująca idea.
- Niezwykle trudna w implementacji i o niejasnej semantyce.
- Problem rombu (diamond problem).
- "*Diamonds aren't inheritance's best friend*" Marilyn Monroe, conf. proc. from 1953, (not 100% sure).

Realizacja w Javie

- Pojedyncze dziedziczenie po klasach.
- Wielodziedziczenie po interfejsach.
- Na razie przyjmijmy, że interfejs to bardzo uproszczona klasa.
- W Javie te dwa rodzaje dziedziczenia są zaznaczane różnymi słowami kluczowymi (**extends** i **implements** odpowiednio).
- Na tym wykładzie: zawsze “dziedziczenie” i często “nadklasa” i “podklasa” w przypadku interfejsów.
- Każda klasa poza *Object* dziedziczy po innej (np. po *Object*).

Przykłady składni dziedziczenia

```
1  class Pracownik extends Osoba{
2          // dziedziczenie po klasie
3   ...}

4  class Samochod implements Pojazd, Towar{
5          // dziedziczenie po kilku interfesjach
6   ...}

7  class Chomik extends Ssak
8                  implements Puchate, DoGlaskania{
9         // dziedziczenie po klasie i kilku interfejsach
10  ...}

11 class Samotnik{
12         // dziedziczenie po klasie Object
13  ...}
```

Java - co klasy dziedziczą

- Metody.
- Atrybuty.
- Także te, które nadklasa sama odziedziczyła.

Przykład z dziedziczeniem

```
1  class A{
2    int iA=1;
3    void infoA(){
4       System.out.println(
5          "Jestem infoA() z klasy A\n"+
6          "  wywolano mnie w obiekcie klasy " +
7          this.getClass().getSimpleName() + "\n" +
8          "  iA="+iA);
9    }
10  }
```

Wyrażenie *this.getClass().getSimpleName()* powoduje wypisanie nazwy klasy obiektu, w którym to wyrażenie jest wyliczane.

Przykład z dziedziczeniem cd.

```
1  class B extends A{
2    int iB=2;
3    void infoB(){
4      infoA();
5      System.out.println(
6        "Jestem infoB() z klasy B\n"+
7        "  wywolano mnie w obiekcie klasy " +
8        this.getClass().getSimpleName() + "\n" +
9        "  iA="+iA + ", iB=" + iB);
10   }
11 }
```

Przykład z dziedziczeniem cd.

```
1  class C extends B{
2    int iC=3;
3    void infoC(){
4      infoA();
5      infoB();
6      System.out.println(
7        "Jestem infoC() z klasy C\n"+
8        "  wywolano mnie w obiekcie klasy " +
9        this.getClass().getSimpleName()+ "\n" +
10       "  iA="+iA + ", iB=" + iB + ", iC=" + iC);
11   }
12 }
```

Przykład z dziedziczeniem cd.

Poniższy fragment

```
1 C c = new C();
2 c.infoC();
```

wypisze ...

Przykład z dziedziczeniem cd.

```
Jestem infoA() z klasy A
  wywolano mnie w obiekcie klasy C
  iA=1
Jestem infoA() z klasy A
  wywolano mnie w obiekcie klasy C
  iA=1
Jestem infoB() z klasy B
  wywolano mnie w obiekcie klasy C
  iA=1, iB=2
Jestem infoC() z klasy C
  wywolano mnie w obiekcie klasy C
  iA=1, iB=2, iC=3
```

Podsumowanie

- Klasa *C* odziedziczyła po swoich przodkach wszystkie atrybuty i metody (podobnie klasa *B*).
- Możemy więc myśleć o obiektach klasy *C* jak o kawałkach tortu składających się z wielu warstw, gdzie każda warstwa odpowiada kolejnej nadklasie ...

Komplikujemy

- Na razie nic szczególnego się nie wydarzyło.
- Skorzystajmy z zasady *podstawialności* i zadeklarujemy obiekt klasy C jako obiekt klasy A
- Dopisujemy więc na końcu naszego programu:

```
1 A a = new C();
2 System.out.println("a.iA=" + a.iA + ", c.iA=" + c.iA);
```

Efekt zmiany

- Oczywiście na wyjściu naszego programu dodatkowo pojawi się poniższy wiersz:

```
a.iA=1, c.iA=1.
```

Złośliwa zmiana

- Wszystkie nazwy atrybutów takie same.
- Oczywiście normalnie unikamy takiej sytuacji.
- Ale w praktyce nie ma jak jej zaradzić (nie znamy dalszych nadklas).
- Co to będzie? (A. Mickiewicz, "Dziady cz. II")

Straszny przykład z dziedziczeniem

```
1  class A{
2    int i=1;
3    void infoA(){
4      System.out.println(
5        "Jestem infoA() z klasy A\n" +
6        "  wywolano mnie w obiekcie klasy " +
7        this.getClass().getSimpleName() + "\n" +
8        "  i z A="+i);
9    }
10 }
```

Straszny przykład z dziedziczeniem cd.

```
1  class B extends A{
2    int i=2;
3    void infoB(){
4      infoA();
5      System.out.println(
6        "Jestem infoB() z klasy B\n" +
7        "  wywolano mnie w obiekcie klasy " +
8        this.getClass().getSimpleName() + "\n" +
9        "  i z A="+
10       ((A) this).i + ", i z B=" + i);   // albo super.i
11   }
12 }
```

Straszny przykład z dziedziczeniem cd.

```
1  class C extends B{
2    int i=3;
3    void infoC(){
4      infoA();
5      infoB();
6      System.out.println(
7        "Jestem infoC() z klasy C\n" +
8        "  wywolano mnie w obiekcie klasy " +
9        this.getClass().getSimpleName()+ "\n" +
10       "  i z A="+ ((A) this).i + ", i z B=" +
11       ((B) this).i + ", i z C=" + i);
12   }
13 }
```

Straszny przykład z dziedziczeniem cd.

Poniższy fragment

```
1 C c = new C();
2 c.infoC();
3
4 A a = new C();
5 System.out.println("a.i=" + a.i + ", c.i=" + c.i);
```

wypisze ...

Straszny przykład z dziedziczeniem cd.

```
Jestem infoA() z klasy A
  wywolano mnie w obiekcie klasy C
  i z A=1
Jestem infoA() z klasy A
  wywolano mnie w obiekcie klasy C
  i z A=1
Jestem infoB() z klasy B
  wywolano mnie w obiekcie klasy C
  i z A=1, i z B=2
Jestem infoC() z klasy C
  wywolano mnie w obiekcie klasy C
  i z A=1, i z B=2, i z C=3
a.i=1, c.i=3
```

Wyjaśnienia do przykładu

- ((A) this) to rzutowanie.
- Nieeleganckie - rezygnujemy z bezpieczeństwa dawanego przez kompilator i silny system typów.
- Gdyby się okazało podczas wykonywania, że typy się nie zgadzają, to zostałby zgłoszony wyjątek.
- Wady dynamicznego sprawdzania typów:
  - spowalnia wykonywanie programu,
  - komunikaty o błędach dostaje niczemu nie winien użytkownik.
- Zawsze unikajmy rzutowań (o ile się da).

Wyjaśnienia do przykładu cd.

- Nadal obiekty *C* mają wszystkie warstwy z nadklas (mimo konfliktu nazw).
- Dostęp do warstwy z bezpośredniej nadklasy: słowo kluczowe **super**.
- Odnosi się do tego obiektu co **this**, ale traktowanego jak obiekt z nadklasy.
- *super.atrybut* oznacza tyle samo co *((Nadklasa) this).atrybut* (ale nie dla metod i dlatego dla metod właśnie będzie bardzo cenny).
- Użyliśmy w programie rzutowania, bo **super** nie pozwala sięgnąć dwa poziomy wyżej.
- *Ogólna uwaga:* poza przykładami ilustrującymi semantykę języka starajmy się nie odwoływać do implementacji innych obiektów, nawet obiektów z nadklas.

Bardzo ważne pytanie

- Co oznacza *a.i*?
- Bardzo ważne pytanie:

  Czy przy odwoływaniu się do składowych obiektu powinien być brany pod uwagę jego typ statyczny (to jest wynikający z deklaracji w programie), czy typ dynamiczny (czyli faktyczny)?

- Dla atrybutów decyduje typ statyczny.
- A dla metod?

Bardzo złośliwa zmiana

- Wszystkie nazwy metod też takie same.
- Normalnie *będziemy dążyć* do takiej sytuacji (choć nie koniecznie w przypadku absolutnie wszystkich metod).
- Co to będzie, co to będzie? (A. Mickiewicz, "Dziady cz. II")

Bardzo straszny przykład II

```
1  class A{
2    int i=1;
3    void info(){
4      System.out.println(
5        "Jestem info() z klasy A\n" +
6        "  wywolano mnie w obiekcie klasy " +
7        this.getClass().getSimpleName() + "\n" +
8        "  i z A>>i = "+i );
9    }
10 }
```

Bardzo straszny przykład II cd.

```
1  class B extends A{
2    int i=2;
3    void info(){
4      super.info();
5      System.out.println(
6        "Jestem info() z klasy B\n" +
7        "  wywolano mnie w obiekcie klasy " +
8        this.getClass().getSimpleName() + "\n" +
9        "  i z A>>i = "+((A) this).i +
10       ", i z B>>i = " + i);   // albo super.i
11     }
12   }
```

Bardzo straszny przykład II cd.

```
1  class C extends B{
2    int i=3;
3    void info(){
4      super.info();
5      System.out.println(
6        "Jestem info() z klasy C\n" +
7        "  wywolano mnie w obiekcie klasy " +
8        this.getClass().getSimpleName()+ "\n" +
9        "  i z A>>i = "+ ((A) this).i +
10       ", i z B>>i = " + ((B) this).i +
11       ", i z C>>i = " + i);
12   }
13 }
```

Bardzo straszny przykład II cd.

Poniższy fragment

```
1 C c = new C();
2 c.info();
3
4 A a = new C();
5 System.out.println("\na.i=" + a.i +
6                    ", c.i=" + c.i + "\n");
7 a.info();
```

wypisze ...

Bardzo straszny przykład II cd.

```
Jestem info() z klasy A
  wywolano mnie w obiekcie klasy C
  i z A»i = 1
Jestem info() z klasy B
  wywolano mnie w obiekcie klasy C
  i z A»i = 1, i z B»i = 2
Jestem info() z klasy C
  wywolano mnie w obiekcie klasy C
  i z A»i = 1, i z B»i = 2, i z C»i = 3

a.i=1, c.i=3
```

Bardzo straszny przykład II cd.

```
Jestem info() z klasy A
  wywolano mnie w obiekcie klasy C
  i z A»i = 1
Jestem info() z klasy B
  wywolano mnie w obiekcie klasy C
  i z A»i = 1, i z B»i = 2
Jestem info() z klasy C
  wywolano mnie w obiekcie klasy C
  i z A»i = 1, i z B»i = 2, i z C»i = 3
```

Wyjaśnienia do przykładu

- Czemu drugie wywołanie metody *info* dało taki sam efekt jak pierwsze?
- Zadziałał polimorfizm - obiekt zareagował we właściwy dla siebie sposób.
- Jak się szuka metod?
- A jak zapewnił to kompilator?
  - Wygenerował kod, który wybrał właściwą metodę.
  - To jest tanie.
  - To jest bezpieczne zwn typy.

Słowo kluczowe **super**

- Znaczenie dla metod:
  - *super.metoda(parametry)*
- Metoda zostanie wywołana na rzecz obiektu **this**, ale wyszukiwanie jej implementacji rozpocznie się od nadklasy klasy, która *(tekstowo)* zawiera wywołanie metody.
- Najczęściej używane w (uproszczonym) znaczeniu - wywołaj metodę z nadklasy.
- Zobaczmy przykład.

super przykład

```
1  class Osoba{
2    private String imie, nazwisko;
3    // ...
4    @Override
5    public String toString(){
6      return "imie = " + imie + ", nazwisko = " +
7             nazwisko;
8    }
9  }
```

super przykład

```
1  class Student extends Osoba{
2     private String nrIndeksu;
3     // Typ String, tak by numer indeksu mógł
4     // zawierać nie tylko cyfry (np. I-123456)
5     //...
6     @Override
7     public String toString(){
8        return super.toString() + ", nr indeksu = " +
9               nrIndeksu;
10    }
11 }
```

Komentarz do przykładu

- Problem: jak sensownie zdefiniować *toString* w klasie *Student*.
- Chcemy wywołać metodę z *Osoby*.
- Umożliwia to **super**.
- Kolejny typowy przykład: konstruktor.

super w konstruktorze

```
1  public Osoba(String imie, String nazwisko){
2    this.imie = imie;
3    this.nazwisko = nazwisko;
4  }
5
6  public Student(String imie, String nazwisko,
7                 String nrIndeksu){
8      super(imie, nazwisko);
9      this.nrIndeksu = nrIndeksu;
10 }
```

Ograniczenia wywoływania konstruktora z nadklasy

- Metodę przez **super** możemy wywołać wszędzie.
- Konstruktor *tylko i wyłącznie* w pierwszej instrukcji.
- Dlaczego?
- Bo treść konstruktora ma sens wykonywać dopiero po zainicjowaniu wyższych warstw.
- A gdy nie napiszemy? Konstruktor bezargumentowy.
- Ta składnia uniemożliwia przechwycenie wyjątków w konstruktorze nadklasy.

Nieintuicyjne użycie **super**

```
1  class StudentStypendysta extends Student{
2    private int stypendium;
3    public StudentStypendysta(String imie,
4      String nazwisko, String nrIndeksu, int stypedium){
5      super(imie, nazwisko, nrIndeksu);
6      this.stypendium = stypedium;
7    }
8    @Override
9    public String toString(){
10     return super.toString() + ", stypendium = " +
11           stypendium;
12   }
13 }
```