Dziedziczenie

Tomasz Borzyszkowski

Podstawy

Zobacz: **Dziedzictwo1.java**
Dziedzictwo2.java

Dziedziczenie jest jedną z podstawowych cech OOP ponieważ umożliwia łatwe implementowanie ***klasyfikacji hierarchicznych***.

W OOP klasę dziedziczoną nazywamy klasą ***nadrzędną*** (*superclass*), natomiast klasę dziedziczącą klasą ***podrzędną*** (*subclass*).

Klasa podrzędna jest wyspecjalizowaną wersją klasy nadrzędnej. Dziedziczy ona wszystkie zmienne instancyjne i metody zdefiniowane w klasie nadrzędnej. Sama dodaje swoje specyficzne elementy. Ogólny schemat definicji klasy *z dziedzictwem*:

```
class nazwaPodklasy extends nazwaNadklasy {
  // ciało klasy
}
```

W Java można dziedziczyć tylko po jednej klasie nadrzędnej.
W C++ można dziedziczyć po wielu klasach. Można za to tworzyć hierarchie klas, w których klasa podrzędna staje się klasą nadrzędną innej klasy podrzędnej, itd.
Klasa nie może być klasą nadrzędną dla samej siebie.

Referencje do klas

Zmienna referencyjna do klasy nadrzędnej może wskazywać na obiekt dowolnej klasy podrzędnej.

Przykład:

```
A obA = new A();
B obB = new B(); obA = obB;
```

Typ zmiennej referencyjnej wyznacza, które elementy wskazywanego obiektu będą dostępne. W powyższym przykładzie zakładamy, że klasa **B** dziedziczy po klasie **A**. Zmienna referencyjna **obA**, wskazująca na obiekt klasy **A**, będzie miała dostęp tylko do tych elementów wskazywanego obiektu, które są dziedziczone z klasy **A**.

Zobacz: **Dziedzictwo3.java**

Zmienna `super`

Zobacz: **SuperDemo.java**
BoxDemo4.java

Zmienna **`super`** służy do odwoływania się do elementów klasy będącej bezpośrednim poprzednikiem klasy obiektu, w którym występuje. **`super`** może być używany na dwa sposoby:

- **Odwołanie do konstruktora** klasy będącej bezpośrednim poprzednikiem klasy ropatrywanego obiektu. Może się to okazać jedyną możliwością inicjalizacji danych prywatnych klasy nadrzędnej. Odwołanie do konstruktora przez **`super`** musi odbywać się w konstruktorze jako pierwsza instrukcja.
- **Odwołanie do zmiennej lub metody** to zastosowanie jest podobne do użycia zmiennej **`this`** z tym, że odwołuje się do klasy nadrzędnej. Często stosuje się, gdy klasa podrzędna definiuje element o takiej samej nazwie jak klasa nadrzędna.

Konstruktory zawsze są wywoływane w kolejności od klasy nadrzędnej do podrzędnej nawet, gdy nie używamy odwołania do konstruktora przez **`super`**.

Zobacz: **KonstrDemo.java**

Nadpisywanie metod

Gdy, w hierarchii klas, metoda w klasie podrzędnej ***posiada tę samą nazwę i sygnaturę*** co metoda w klasie nadrzędnej, wówczas mówimy, że metoda z klasy podrzędnej ***nadpisuje*** metodę klasy nadrzędnej.

Wywołanie nadpisanej metody z klasy podrzędnej zawsze spowoduje wykonanie kodu metody z klasy podrzędnej. Wersja metody z klasy nadrzędnej będzie przesłonięta wersją z klasy podrzędnej, dostępną wciąż przez zmienną **`super`**.

Zobacz: **Overriding1.java**
Overriding2.java

Z nadpisywaniem metod mamy do czynienia tylko wtedy, gdy nazwy i sygnatury dwóch metod są identyczne. Jeżeli metody mają takie same nazwy lecz różne sygnatury, to mamy do czynienia z przeciążaniem metod, omówionym już wcześniej.

Zobacz: **Overriding3.java**

Dynamiczne wiązanie metod

Nadpisywanie metod tak jak je przedstawiliśmy nie jest niczym szczególnym. Dopiero z wraz mechanizmem ***dynamicznego wiązania metod***, tworzy podstawę dla ***polimorfizmu***. Dynamiczne wiązanie metod polega na tym, że dopiero w *czasie wykonywania programu* można zdecydować, która wersja nadpisanej metody będzie uruchomiona (a nie już w czasie kompilacji).

Zmienne referencyjne typu nadklasy mogą wskazywać również na obiekty klasy podrzędnej. Java wykorzystuje ten fakt do rozwiązywania odwołań do nadpisanych metod w czasie wykonania w następujący sposób:

> *Wywołanie nadpisanej metody przez referencję typu nadklasy jest* ***wyznaczone przez typ obiektu*** *wskazywanego przez referencję w czasie wykonania*.

Prezentowany sposób nadpisywania metod bardzo przypomina tzw. funkcje wirtualne języka C++.

Zobacz: **DynMetod.java**

Zastosowanie polimorfizmu

Polimorfizm w OOP jest ważnym mechanizmem ponieważ ułatwia definiowane hierarchii klas. Możemy definiować klasy abstrakcyjne, definiujące cechy wspólne, a następnie w klasach potomnych doprecyzować definicje metod z klas abstrakcyjnych dla poszczególnych przypadków.

Przykład:

W pliku **Figury.java** *pokazujemy jak przy pomocy nadpisywania metod i polimorfizmu można zdefiniować hierarchię klas opisujących figury geometryczne.*

Ciekawe wywołania polimorficzne można uzyskać stosując, prócz nadpisywania metod, zmienne **`this`** i **`super`**.
Przykład można znaleźć w pliku **DynWia.java**.

Zobacz: **Figury.java**

Klasy abstrakcyjne

Zobacz: **SimpleAbs.java**
BoxDemo4.java

Zdarzają się sytuacje, w których chcemy zdefiniować abstrakcyjną klasę nadrzędną, deklarującą *strukturę abstrakcji* lecz bez implementowania wszystkich metod. Klasa taka powinna nadawać kształt (sygnaturę) metod w klasach potomnych lecz sama o niczym nie przesądzać.

Metody takie możemy tworzyć przez poprzedzenie ich nagłówka słowem kluczowym **`abstract`**. Klasy takie nie posiadają implementacji, tj. składają się tylko z nagłówka. Postać metody:

```
abstract typ nazwa(lista-parametrów);
```

Każda klasa, która posiada chociaż jedną metodę abstrakcyjną, musi także być zadeklarowana jako abstrakcyjna (**`abstract`** przed słowem **`class`**).

Nie można tworzyć instancji klas abstrakcyjnych, chociaż można deklarować zmienne referencyjne klas abstrakcyjnych i wskazywać nimi obiekty klas pochodnych.

Słowo `final`

Zobacz: **FinalMethod.java**
FinalClass.java

Słowo kluczowe **`final`** posiada trzy zastosowania:

- ***Niezmienne* zmienne instancyjne** czyli stałe; omówiliśmy już to zagadnienie wcześniej
- **Metody *nienadpisywalne*** metody posiadające słowo kluczowe **`final`** w nagłówku *nie mogą być nadpisane* w klasach pochodnych; stąd są w pewien sposób *niezmienne w klasach pochodnych*
- **Klasy *nie dające się dziedziczyć*** klasy takie nie mogą być dziedziczone do żadnych klas potomnych; wszystkie metody takiej klasy niejawnie są zadeklarowane jako **`final`**; jak można się domyślać błędem jest deklarowanie klasy jako **`abstract`** i **`final`** jednocześnie

Kompilatory Javy często, dla wygenerowania efektywniejszego kodu, wpisują kod metody finalnej w miejsce jej wywołania. Oszczędza to czas potrzebny na wywołanie metody. Taki sposób kompilacji nazywamy ***wczesnym wiązaniem***, tj. wiązaniem kodu metody z jej wywołaniem w trakcie kompilacji.

Klasa `Object`

Klasa **`Object`** jest specjalną klasą języka Java. Wszystkie klasy języka Java są jej podklasami. Oznacza to, że zmienna referencyjna typu **`Object`** może wskazywać na obiekt dowolnej klasy.

Klasa **`Object`** definiuje następujące metody, które są dostępne we wszystkich klasach Javy:

- **`Object clone()`** tworzy obiekt taki sam, jak obiekt wywołujący
- **`boolean equals(Object o)`** sprawdza równość obiektów wywołującego i **`o`**
- **`void finalize()`** wywoływana przed odzyskaniem pamięci obiektu przez zbieracz śmieci
- **`Class getClass()`** oddaje klasę obiektu wywołującego
- **`int hashCode()`** oddaje pozycję obiektu wywołującego w tablicy mieszanej
- **`String toString()`** oddaje napis opisujący obiekt wywołujący

Dokumentacja klas

W skład SDK wchodzi program **`javadoc`**, służący do generowania dokumentacji klasy na podstawie pliku źródłowego oraz specjalnych znaczników:

- **`/** .. */`** komentarz będący jednocześnie dokumentacją komentowanego elementu
- **`@autor`** autor klasy lub komentowanego elementu
- **`{@link}`** link do dodatkowej informacji
- **`@param`** parametry metody
- **`@return`** opis wartości zwracanej przez metodę
- **`@see`** link do tematu pokrewnego
- **`@exception`** wyjątki generowane przez metodę
- **`@version`** wersja klasy

Zobacz: **Sortowanie.java**

Wypróbuj: **`javadoc Sortowanie.java`**