Pakiety i interfejsy

Tomasz Borzyszkowski

Pakiety podstawy

W dotychczasowych przykładach nazwy klas musiały pochodzić z **jednej przestrzeni nazw**, tj. być **niepowtarzalne** tak, by nie doprowadzić do **kolizji nazw**. Dbanie o niepowtarzalność nazw stwarzało by wiele problemów w projektach zespołowych, gdyż zmuszało by programistów do uzgadniania nazw klas ze wszystkimi innymi programistami projektu.

Java oferuje mechanizm pozwalający na **podział przestrzeni nazw** klas na niezależne przestrzenie o pełnym zakresie nazw każda. Tym mechanizmem jest **podział projektu na pakiety**. Pakiet składa się z definicji klas o nazwach niezależnych od nazw klas z innych pakietów oraz daje możliwość kontroli dostępu do klas pakietu z innych klas.

Np. możemy utworzyć pakiet zawierający klasę `List` nie martwiąc się tym, że może kolidować z innymi klasami o takiej samej nazwie umieszczonych w innych pakietach.

Pakiety definicje

Aby utworzyć pakiet należy w pierwszej linii pliku ze źródłami klas umieścić komendę:

```
package nazwa;
```

nazwa jest nazwą nowej przestrzeni nazw klas (nazwą pakietu).

Ominęcie tej **komendy** na początku pliku (*czyli tak jak było do tej pory*) powoduje, że wszystkie klasy zdefiniowane w pliku będą umieszczone w domyślnym pakiecie bez nazwy.

Umieszczenie komendy `package` z tą samą nazwą pakietu **w kilku plikach** z definicjami klas spowoduje umieszczenie klas zdefiniowanych we wszystkich tych plikach w jednym pakiecie.

Możemy tworzyć hierarchie pakietów, oddzielając ich nazwy kropką:

```
package nazwa1.nazwa2.nazwa3;
```

Powyższy pakiet musi być przechowywany w katalogu **`nazwa1/nazwa2/nazwa3`**. Nie można zmieniać nazwy pakietu bez zmiany nazw katalogów, w których klasy są przechowywane.

Zmienna systemowa `CLASSPATH`

`CLASSPATH` jest zmienną systemową, która określa położenie klas w systemowym drzewie katalogów. Zmienna ta jest używana do odnajdywania definicji klas zarówno przez kompilator jak i maszynę wirtualną.

Zobacz: **PackTest.java**
oraz: **setCLASSPATH**

Przykład:

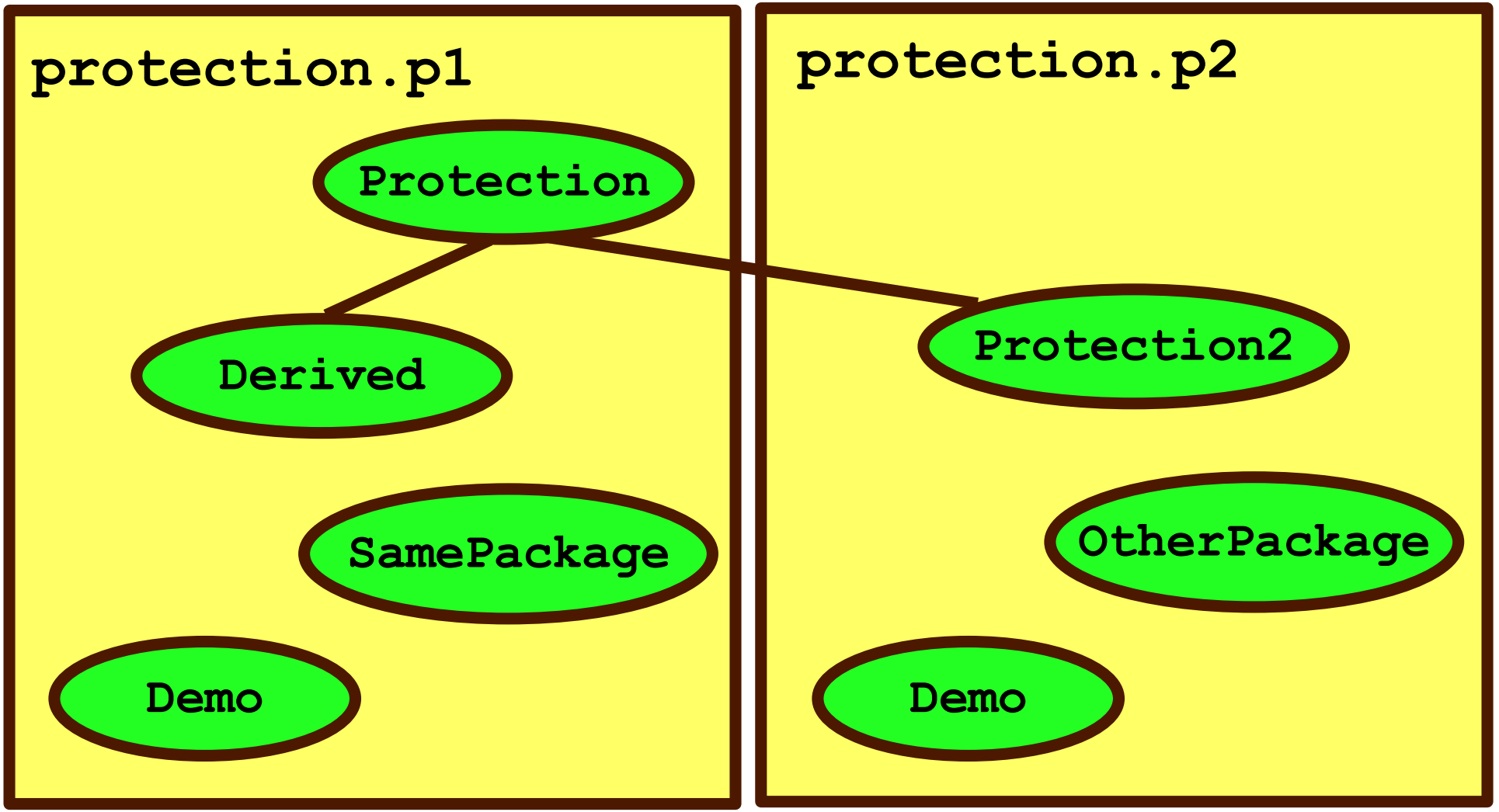
Definiujemy klasę **`PackTest`** *umieszczoną w pakiecie* **`test`**.
Ponieważ hierarchia pakietów musi odpowiadać hierarchii katalogów, musimy utworzyć katalog **`test`** *i umieścić w nim plik* **`PackTest.java`**.
Przechodzimy do katalogu test i kompilujemy plik **`PackTest.java`**.
W wyniku otrzymujemy plik **`PackTest.class`** *w katalogu* **`test`**.
Próba uruchomienia **`PackTest`** *z katalogu* **`test`** *lub zawierającego* **`test`** *spowoduje błąd.*
Należy dodać katalog zawierający **`test`** *do* **`CLASSPATH`** *i uruchomić klasę* **`test.PackTest`** *z dowolnego miejsca.*

Zobacz: **MyPack/ccountBalance.java**

Kontrola dostępu

Zakres	`private`	*nic*	`protected`	`public`
Klasa	TAK	TAK	TAK	TAK
Podklasy w pakiecie	NIE	TAK	TAK	TAK
Inne klasy w pakiecie	NIE	TAK	TAK	TAK
Podklasy w innym pakiecie	NIE	NIE	TAK	TAK
Inne klasy w innym pakiecie	NIE	NIE	NIE	TAK

Kontrola dostępu przykład



Zobacz katalog: **protection**

Importowanie pakietów

Wszystkie klasy języka Java są przechowywane w pakietach.
Ich użycie wymaga podawania ***pełnej, pakietowej ścieżki dostępu*** do klasy. Dla uproszczenia zapisu nazw klas w pakietach zewnętrzych można używać, i zwykle tak się robi, komendy:

```
import pgk1[.pkg2].(nazwa_klasy | *);
```

Po zaimportowaniu klasa jest widoczna i można się do niej odwoływać bezpośrednio.
Użycie gwiazdki może wydłużyć czas kompilacji. Szczególnie gdy, importujemy kilka dużych pakietów. Dlatego dobrym zwyczajem jest raczej wskazanie konkretnej klasy niż importowanie całego pakietu. Z drugiej strony **użycie gwiazdki nie ma wpływu na czas działania** programu ani na wielkość kodu wynikowego.

Istnienie klasy o takiej samej nazwie w dwóch zaimportowanych pakietach nie spowoduje błędu póki w kodzie nie wystąpi odwołanie do tej klasy. Trzeba wówczas użyć pełnej ścieżki dostępu do klasy.

Zobacz: MyPack i plik: TestBalance.java

Interfejsy definicja

Interfejs specyfikuje **co** klasa musi implementować, jednak nie specyfikuje **jak**. Interfejsy są syntaktycznie podobne do klas, nie posiadają natomiast zmiennych instancyjnych, a ich metody nie posiadają ciała.

Klasa może implementować dowolną liczbę interfejsów oraz dowolna liczba klas może implementować jeden interfejs. Interfejs nie zakłada sposobu implementacji swoich metod.

```
access interface nazwa{
   typ zmienna_finalna1 = wartość;
   typ nazwa_metody1(lista_param);
   ...
}
```

Jeżeli ***access*** jest **public**, to interfejs jest dostępny publicznie i jego elementy są także niejawnie publiczne.
Jeżeli **access** nie występuje, to interfejs jest dostępny tylko wewnątrz pakietu. Zmienne występujące w interfejsie są niejawnie **final** i **static**.

Interfejsy implementacja

Implementacja interfejsu polega na dodaniu słowa kluczowego **`implements`** z nazwą interfejsu do nagłówka klasy oraz na dodaniu implementacji metod zadeklarowanych w interfejsie. Postać ogólna:

```
access class klasa [extends superklasa]
   [implements interfejs [,interfejs]]  {
   // ciało klasy
}
```

Tutaj ***`access`*** jest **`public`** lub nie występuje wcale. Jeżeli klasa implementuje więcej niż jeden interfejs, są one rozdzielane przecinkiem. Jeżeli dwa implementowane interfejsy deklarują tę samą metodę, to ta sama metoda będzie używana przez klientów obu interfejsów. Metody implementujące metody interfejsu muszą być zadeklarowane jako **`public`**.

Interfejsów można używać jak typów, zamiast klas. Wówczas odpowienia implementacja interfejsu zostanie wybrana w trakcie wykonania programu. Interfejsy mogą być także implementowane przez klasy abstrakcyjne.

Zobacz: **Callback.java**

Zastosowanie interfejsów

W poprzednich rozdziałach przezentowaliśmy kilka implementacji stosów. Istnieje wiele możliwych implementacji idei stosu. Np. można implementować stosy o stałej wielkości lub zmiennej, wybierając wewnętrzną implementację w postacji tablic, list, drzew binarnych, itd.

Wspólną cechą wszystkich takich stosów będzie to, że możemy przy pomocy metody `push(e)` dodawać element `e` na wierzchołek stosu, natomiast za pomocą metody `pop()` zdejmować element ze stosu.

Wspólne cechy stosów liczb całkowitych można wyrazić w postaci następującego interfejsu: **patrz IntStack.java**

Następnie implementujemy powyższy interfejs na dwa sposoby:

- Stosy o stałej wielkości **patrz: FixedStack.java**
- Stosy o zmiennej wielkości **patrz: DynStack.java**

Interfejsy zmienne i rozszerzanie

Istnieje możliwość wykorzystania interfejsu do importowania stałych do wszystkich klas implementujących interfejs. Wewnątrz interfejsu stałym powinna zostać nadana odpowiednia wartość. Wszystkie stałe zaimportowane do klasy w taki sposób są niejawnie traktowane jak zmienne **`final`**.

Zobacz: **FinalIF.java**

Interfejs może być rozszerzeniem innego interfejsu, podobnie jak jedna klasa może rozszerzać inna klasę. Składnia takiego rozszerzenia jest także podobna do rozszerzania klas. Klasa implementujaca interfejs musi implementować wszystkie metody tego interfejsu oraz wszystkie metody dziedziczone przez ten interfejs.

Zobacz: **ExtendIF.java**