`java.util.*` :Kolekcje

Tomasz Borzyszkowski

Wstęp

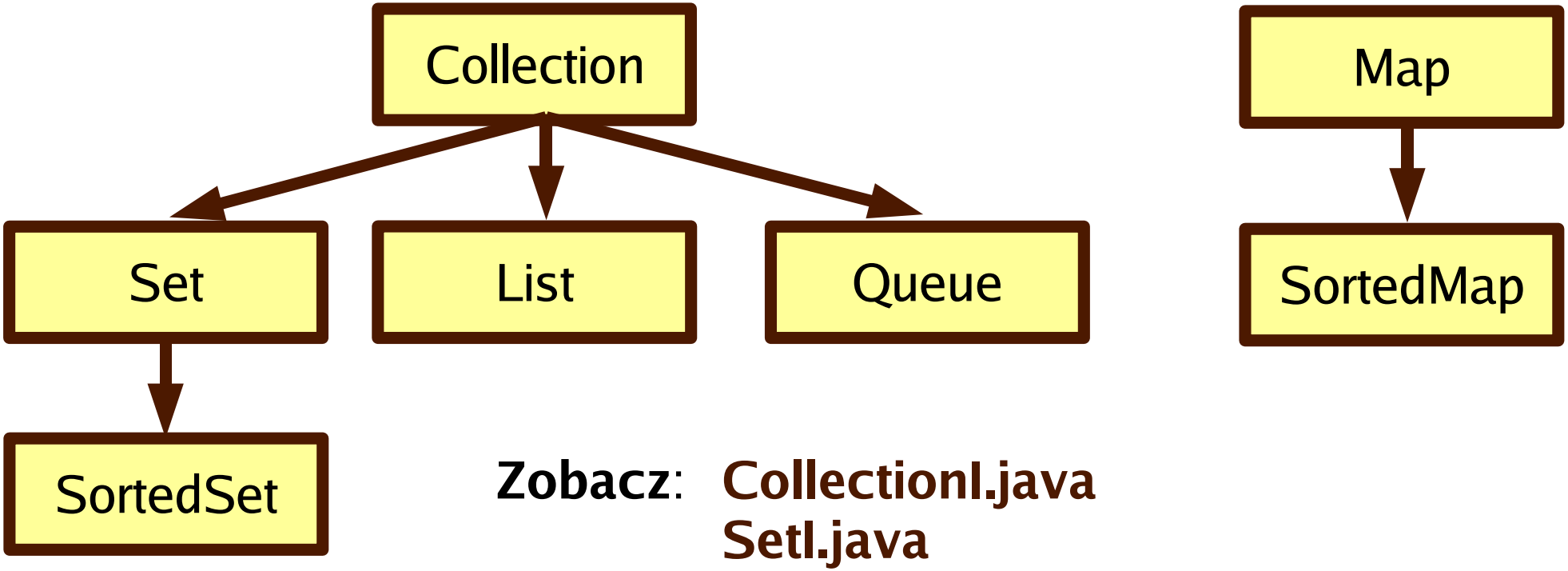
Kolekcje w Java dają programiście pewien standardowy sposób radzenia sobie z przetwarzaniem grup obiektów. Implementacja kolekcji w Java składa się z następujących składowych:

- **Interfejsów**: definiujących abstrakcyjne właściwości i operacje kolekcji, w oderwaniu od konkretnych implementacji
- **Implementacji**: klasy będące implementacjami odpowiednich interfejsów
- **Algorytmów**: metod pozwalających na efektywne przetwarzanie kolekcji; np.: wyszukiwanie czy sortowanie. Algorytmy te zwykle są *polimorficzne*, tj. są zdefiniowane dla pewnego rodzaju kolekcji a nie tylko dla wybranej klasy

Podstawowe cechy kolekcji to: wysoka efektywność kodu oraz to, że kolekcje różnych typów posiadają podobną i prostą obsługę. Kolekcje implementują także interfejs `Iterator`, ułatwiający i standaryzujący dostęp do kolejnych elementów kolekcji.

Interfejsy

Hierarchia podstawowych interfejsów wykorzystywanych w kolekcjach:



Zobacz: **CollectionI.java**
SetI.java
ListI.java
QueueI.java
SortedSetI.java

Typy parametryczne

Z kolekcjami w Java 5.0 związane są tzw. typy parametryczne. Pozwalają one parametryzować definicje klas i interfejsów typami, które będą określone później, np. w trakcie użycia klasy czy interfejsu.

Zobacz: **Stack_1.java**
Stack_1G.java

Zauważ, że:

- typ parametryczny T wprowadzamy za nazwą definiowanego typu, natomiast wewnątrz definicji klasy używamy go jak zwykłego typu
- typ **`Stack<Integer>`** jest typem stosów elementów typu **`Integer`** i nie może zawierać elementów innych typów (np.: **`String`**)
- ponieważ kompilator Javy *wie*, że wszystkie elementy kolekcji są jednorodne, więc nie są wymagane jawne rzutowania, takie jak:

```
String napis = (String)s.pop();
```

Klasa `ArrayList`

Klasa ta implementuje interfejs **`List`**. W odróżnieniu od zwykłych tablic obiekty tego typu mogą dynamicznie się powiększać by pomieścić większą liczbę elementów – są tablicami o zmiennej długości.

Zobacz: **ArrayListDemo.java**
ArrayListDemoG.java

```
ArrayList()
ArrayList(int capacity)
ArrayList(Collection<? extends E> c)
```

Czasem zachodzi potrzeba przepisania kolekcji typu **`ArrayList`** na zwykłą tablicę. Można to zrealizować za pomocą metody **`toArray()`**.

Zobacz: **ArrayListToArray.java**
ArrayListToArrayG.java

Klasa `LinkedList`

Klasa ta także implementuje interfejs **`List`**. Jest ona realizacją koncepcji *listy dowiązaniowej*.

```
LinkedList()
LinkedList(Collection<? extends E> c)
```

Klasa definiuje typowe metody związane z obsługą list, takie jak: **`addFirst(E o)`**, **`addLast(E o)`**, **`getFirst()`**, **`getLast()`**, **`removeFirst()`**, **`removeLast()`**.

Zobacz: **LinkedListDemo.java**
LinkedListDemoG.java

Klasa `HashSet` i `TreeSet`

Klasa **`HashSet`** realizuje koncepcję tablicy z *kodowaniem mieszanym*. W zwykłych tablicach znalezienie elementu często wiąże się z przejrzeniem sekwencyjnym wszystkich elementów tablicy.
W *tablicach hashowych* mamy do dyspozycji specjalną funkcję, która każdemu elementowi przypisuje jego indeks. Dlatego nawet standardowe operacje (wyszukiwanie, usuwanie, dodawanie, ...) zwykle zajmują stały czas, nawet dla dużych tablic.
Zwykle współczynnik wypełnienia tablicy jest 75% (patrz konstruktory).

Kolejną klasą jest klasa **`TreeSet`**. Realizuje ona koncepcję posortowanego drzewa binarnego. Nadaje się doskonale do przechowywania dużej liczby posortowanych informacji, do których chcemy mieć szybki dostęp.

Zobacz: **HashSetDemo.java**
TreeSetDemo.java

Iteratory

Zobacz: **IteratorDemo.java**
ForLoop.java
IteratorDemo_2.java

Zanim uzyskamy dostęp do kolekcji za pomocą iteratora, powinniśmy utworzyć obiekt *iteratora*. Służy do tego metoda **`iterator()`** dostępna w każdej klasie. Ogólnie by użyć iteratora do *przechodzenia po* elementach kolekcji trzeba:

- Utworzyć iterator wskazujący na początek kolekcji, przez wywołanie metody **`iterator()`**
- Utworzyć pętlę z warunkiem przejścia **`hasNext()`**
- Wewnątrz pętli możemy otrzymać bieżący element przez wywołanie metody **`next()`**

Dla kolekcji implementujących interfejs **`List`** możemy utworzyć iteratora za pomocą wywołania metody **`listIterator()`.**
W Java 5.0 dodano możliwość iterowania po kolekcjach i tablicach bez konieczności tworzenia iteratora – szczegóły w przykładach.

Klasy typu *Map*

Zobacz: **HashMapDemo.java**
TreeMapDemo.java

Obiekt klasy *Map* przechowuje związki pomiędzy *kluczem* a *wartością*. Znając klucz, można łatwo znaleźć związaną z nim wartość. Zarówno klucze jak i wartości są obiektami. O ile klucze muszą być unikalne, wartości mogą się powtarzać.

Przykładem klasy typu *Map* jest klasa **`HashMap`**. Podobnie jak **`HashSet`** zapewnia *prawie* stały czas dostępu do przechowywanych elementów – nawet dla dużych kolekcji. Klasa ta nie grawantuje żadnej kolejności przechowywanych elementów, tj. Kolejność dodawania elementów do kolekcji nie zawsze jest taka jak kolejność otrzymana za pomocą odp. iteratora.

Kolejnym przykładem jest klasa **`TreeMap`**. Klasa ta umożliwia przechowywanie elementów w zadanym porządku przy bardzo efektywnej metodzie dostępu do elementów.

Comparators

Zobacz: **MyComp.java**
TreeMapDemo2.java

Klasy **`TreeSet`** i **`TreeMap`** przechowują elementy w pewnym ustalonym, *naturalnym* porządku. Obie klasy posiadają konstruktor, który jako argument pobiera obiekt klasy **`Comparator`**. Klasa ta umożliwia przekazanie do klas **`TreeSet`** i **`TreeMap`** nowego porządku.

Interfejs **`Comparator`** definiuje dwie metody:

- **`int compare(Object obj1, Object obj2)`**: definiuje porządek na obiektach; oddaje:
  - 0, gdy **`obj1=obj2`**,
  - 1, gdy **`obj1>obj2`** i
  - -1, gdy **`obj1<obj2`**; metoda może także oddać wyjątek
  - **`ClassCastException`**, gdy typy porównywanych obiektów są nieporównywalne
- **`boolean equals(Object obj)`**: metoda oddaje **`true`**, gdy **`obj`** i obiekt wywołujący są klasy **`Comparator`** i używają tego samego porządku, w przeciwnym przypadku oddaje **`false`**.

Dostępne implementacje

W Java dostępne są następujące implementacje kolekcji:

Interfejsy		Implementacje: Tablica hashowa	Tablica o zmiennej długości	Drzewo wyważone	Lista wiązana	Tablica hashowa + Lista wiązana
	Set	`HashSet`		`TreeSet`		`LinkedHashSet`
	List		`ArrayList`		`LinkedList`	
	Map	`HashMap`		`TreeMap`		`LinkedHashMap`

Algorytmy i tablice

W Java zdefiniowano wiele algorytmów związanych z przetwarzaniem kolekcji. Algorytmy te są zdefiniowane jako statyczne metody klasy **`Collections`**.

Realizują one takie operacje jak: sortowanie, wyszukiwanie elementów, kopiowanie kolekcji, czy synchronizowanie kolekcji na potrzeby wątków.

Zobacz: **Sort.java**
Anagram2.java + **dictionary.txt**

Już w Java 2 do pakietu **`java.util`** dodano klasę **`Arrays`**.

Klasa ta dostarcza wielu metod statycznych, usprawniających pracę z tablicami. Chociaż klasa ta nie należy do *kolekcji*, pozwala wypełnić lukę pomiędzy kolekcjami i tablicami.

Zobacz: **ArraysDemo.java**

Klasy Javy 1.1

Przed wprowadzeniem Javy 1.2, Java nie posiadała kolekcji. W zamian udostępniono kilka klas służących do przechowywania obiektów.
W Java 1.2 zachowano je, przebudowując je jednak tak, by współpracowały z kolekcjami.
Klasy te są synchronizowane – w odróżnieniu od klas *kolekcji*.
Przykładowe *stare* klasy:

- **`Vector`** i podklasa **`Stack`** podobne do **`ArrayList`**
- **`Dictionary`** podobna do **`Map`**
- **`Hashtable`** podobna do **`HashMap`**

Zobacz: **VectorDemo.java**
StackDemo.java
Phonebook.java