Programowanie w języku Java - Kolekcje (kontenery)

mgr inż. Maciej Lasota <m.lasota@tu.kielce.pl>

Version 1.0, 16-04-2017

Spis treści

Java Collections Framework (JCF) 1
Listy 3
Kolejki 4
Zbiory 5
Mapy 6
Iteratory 7
Operacje na tablicach 8
Bibliografia 9

Java Collections Framework (JCF)

Kolekcja jest obiektem, który *grupuje elementy* danych (inne obiekty) i pozwala traktować je jak jeden **zestaw danych**, umożliwiając jednocześnie wykonywanie operacji na zestawie danych np. *dodawania i usuwania oraz przeglądania elementów zestawu*. W pakiecie `java.util`, zdefiniowano narzędzia, służące do tworzenia i posługiwania się rożnymi rodzajami kolekcji.

Każda z tych **abstrakcyjnych struktur danych** ma pewne właściwości, które wyróżniają ją od innych struktur danych. Mówimy, że są to abstrakcyjne struktury danych, bowiem w opisie tych właściwości nie przesądza się o tym w jaki sposób konkretnie je zrealizować.

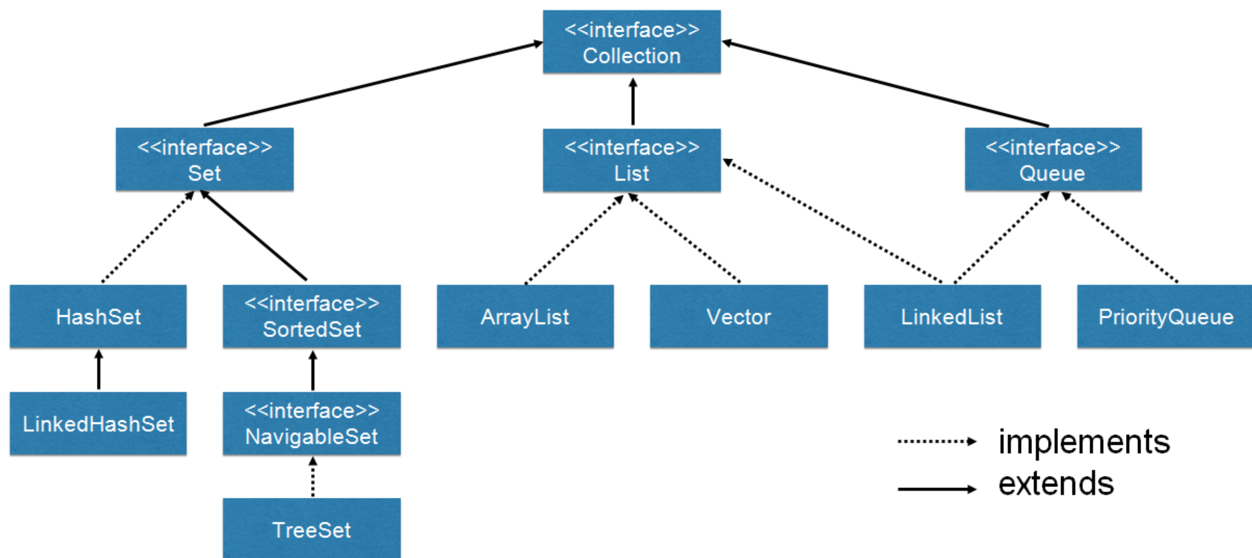
Abstrakcyjne właściwości struktur danych opisywane są przez **interfejsy**, a konkretne realizacje inaczej **implementacje** tych właściwości znajdujemy w konkretnych klasach.

Architektura kolekcji (Collections Framework), składa się z:

- interfejsów,
- implementacji,
- algorytmów.

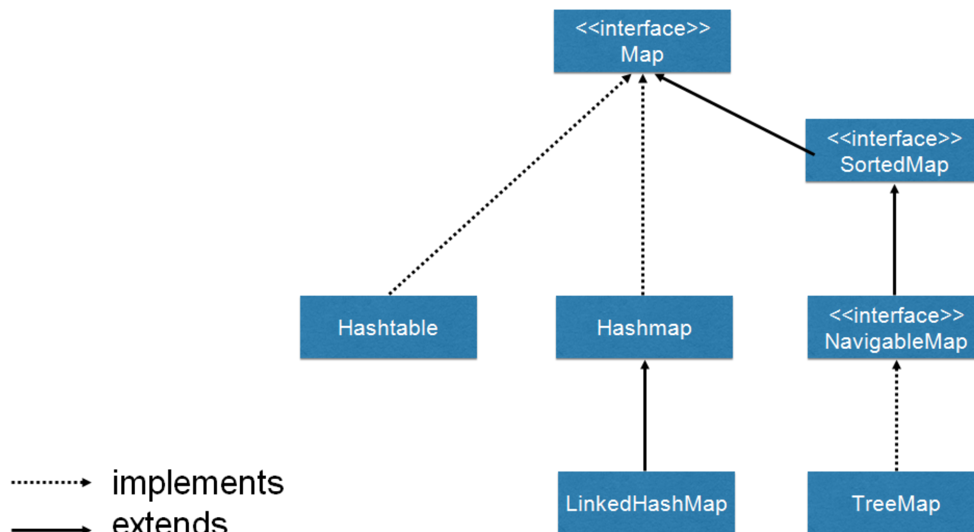
Kolekcje w Javie możemy podzielić na:

1. **Listy** *(ang. Lists)* - klasy implementujące interfejs `java.util.List`,
2. **Kolejki** *(ang. Queues)* - klasy implementujące interfejs `java.util.Queue`,
3. **Zbiory** *(ang. Sets)* - klasy implementujące interfejs `java.util.Set`,
4. **Mapy** *(ang. Maps)* - klasy implementujące interfejs `java.util.Map`.



Rysunek 1: Collection Interface.

Kolekcje z biblioteki standardowej Java SE to wszystkie klasy implementujące interfejs `java.util.Collection` (jedynym wyjątkiem są **Mapy** które nie implementują interfejsu Collection lecz interfejs **Map**, jednak są zaliczane do standardowych kolekcji języka Java).



Rysunek 2: Map Interface.

Interfejs `java.util.Collection` umożliwia podstawowe operacje na kolekcji:

- `.add()` - dodaje element do kolekcji,
- `.remove()` - usuwa pojedynczy element z kolekcji, o ile istnieje,

- `.size()` - zwraca liczbę elementów w kolekcji,
- `.iterator()` - zwraca iterator kolekcji,
- `.toArray()` - zwraca tablicę elementów zawartych w kolekcji.

Java Collections Framework (JCF) jest niezwykle użyteczną składową środowiska Javy. Mamy do dyspozycji wiele gotowych, efektywnych, klas i metod, pozwalających łatwo rozwiązywać wiele problemów związanych z reprezentacją w programie bardziej zaawansowanych struktur danych i operowaniem na nich.

Listy

Listy to kolekcje uporządkowane, w których szczególnie istotna jest kolejność elementów. Umożliwiają one operowanie na swych elementach z użyciem **indeksów**, np. *pobranie elementu znajdującego się na konkretnej pozycji listy*.

Wśród gotowych implementacji list w JCF są dostępne następujące klasy:

- **ArrayList** - lista tablicowa,
- **LinkedList** - lista liniowa z podwójnymi dowiązaniami.

Implementacja **tablicowa** polega na realizacji listy w postaci *tablicy* z dynamicznie (w miarę potrzeby) zwiększającymi się rozmiarami. Elementy listy są zapisywane jako elementy takiej tablicy. Ponieważ tablice w Javie mają określone (niezmienne po utworzeniu) rozmiary utworzenie listy tablicowej wymaga alokacji tablicy z jakimś zadaną rozmiarem. Jest on specyfikowany przez `initialCapacity` *(domyślnie 10)*, który to parametr możemy podać w konstruktorze `ArrayList`, jeśli inicjalna pojemność nam nie odpowiada. Przy dodawaniu elementów do listy sprawdzane jest czy pojemność tablicy jest wystarczająca, jeśli nie to rozmiar tablicy jest *zwiększany*, Służy temu metoda `ensureCapacity(minCapacity)`, którą zresztą możemy wywołać sami, aby w trakcie działania programu zapewnić podaną jako `minCapacity` pojemność listy.

Implementacja **liniowa** z podwójnymi dowiązaniami umieszcza dane w strukturach danych nazwiemy je wiązaniami (link) - które zawierają wskaźniki do poprzedniego i następnego elementu listy *(dlatego dowiązania są "podwójne", niejako dwukierunkowe)*. Zatem elementy listy, które z punktu widzenia programisty są elementami umieszczanych danych *(np. nazwisk lub jakichś innych obiektów)*, technicznie są "linkami", zawierającymi nie tylko dane, ale również wskaźniki na następny i poprzedni element na liście. Początek listy dowiązaniowej, zwany głową lub wartownikiem zawiera wskazanie na pierwszy element listy (`null`, jeśli lista jest pusta).

Interfejs `java.util.List` umożliwia dodatkowe operacje na liście:

- `.indexOf()` - zwraca indeks pierwszego wystąpienia określonego elementu w liście,
- `.lastIndexOf()` - zwraca indeks ostatniego wystąpienia określonego elementu w liście,
- `.set()` - zastępuje element na wskazanej pozycji nowym elementem.

Przykład 1. Przykład użycia ArrayList

```
import java.util.ArrayList;

public class ArrayListExample {

  public static void main(String[] args) {

    ArrayList arrayList = new ArrayList();

    arrayList.add("1");
    arrayList.add("2");
    arrayList.add("3");

    System.out.println("Getting elements of ArrayList");
    System.out.println(arrayList.get(0));
    System.out.println(arrayList.get(1));
    System.out.println(arrayList.get(2));
  }
}
```

Kolejki

Kolejki zaprojektowane są do przechowywania elementów które powinny być przetwarzane kolejno, zgodnie z pewnym porządkiem. Kolejka jest sekwencją elementów, na której operacje wstawiania, pobierania i usuwania elementów są możliwe tylko w określonym porządku. Kolejki mogą być nieograniczone (nie ma limitu na miejsce) lub ograniczone (mieszczące jakąś określoną liczbę elementów).

W JCF wyróżniamy następujące rodzaj kolejek:

- **Kolejki FIFO** *(first in - first out)*,
- **Kolejki LIFO** *(last-in - first-out, inaczej zwane stosami - ang. stack)*,
- **Kolejki z priorytetami**.

Oprócz zwykłych kolejek w Javie dostępne są też **kolejki podwójne** *(ang. deque)* będące liniową sekwencją elementów, na której operacje wstawiania, pobierania i usuwania elementów są możliwe na obu końcach sekwencji.

Wśród gotowych implementacji kolejek w JCF są dostępne następujące klasy:

- **LinkedList** - lista, która jest jednocześnie kolejką podwójną (bez ograniczeń na miejsce),
- **ArrayDeque** - kolejka podwójna, zrealizowana jako rozszerzalna tablica,
- **PriorityQueue** - kolejka z priorytetami.

Interfejs `java.util.Queue` umożliwia dodatkowe operacje na kolejce:

- `.add()` lub `.offer()` - dodawanie elementu do kolejki,

- `.remove()` lub `.poll()` - usuwanie elementu z kolejki,
- `.element()` lub `.peek()` - zwraca element z początku kolejki.

Przykład 2. Przykład użycia LinkedList

```
import java.util.LinkedList;

public class LinkedListExample {

  public static void main(String[] args) {

    LinkedList lList = new LinkedList();

    lList.add("1");
    lList.add("2");
    lList.add("3");
    lList.add("4");
    lList.add("5");

    System.out.println("LinkedList contains : " + lList);
  }
}
```

Zbiory

Zbiory charakteryzują się tym, że nie mogą zawierać **duplikatów**, tj. *nie mogą zawierać dwu takich samych elementów*. Zbiory wykorzystujemy wszędzie tam, gdzie istotny jest sam fakt należenia bądź nie należenia elementu do zbioru oraz tam gdzie chodzi o proste przechowywanie pewnej grupy elementów.

Wśród gotowych implementacji zbiorów w JCF są dostępne następujące klasy:

- **HashSet** - zbiór w którym wyszukiwanie elementów realizowane jest z użyciem tablicy mieszającej,
- **TreeSet** - zbiór w którym wyszukiwanie elementów realizowane jest z użyciem drzewa czerwono-czarnego.

Tablica mieszająca *(hashtable)* jest strukturą danych specjalnie przystosowaną do szybkiego odnajdywania elementów. Dla każdego elementu danych wyliczany jest kod numeryczny (liczba całkowita) nazywany **kodem mieszania** *(hashcode)*, na podstawie którego obliczany jest indeks w tablicy, pod którym będzie umieszczony dany element. W Javie można wyliczyć kod mieszania dla każdego obiektu za pomocą zastosowania metody `hashCode()`.

Z punktu widzenia operowania na zbiorach mamy do dyspozycji metody interfejsu `java.util.Set` (które są takie same jak omówione wcześniej metody interfejsu **Collection**). Jedyne uszczegółowienie polega na tym, że w przypadku zbiorów, metody dodające elementy do zbiorów zwracają wartość `false`, jeśli dodawane elementy już w zbiorze występują.

Przykład 3. Przykład użycia HashSet

```
import java.util.HashSet;

public class HashSetExample {

  public static void main(String[] args) {

    HashSet hSet = new HashSet();

    hSet.add(new Integer("1"));
    hSet.add(new Integer("2"));
    hSet.add(new Integer("3"));

    System.out.println("HashSet contains.." + hSet);
  }
}
```

Mapy

Mapy w języku Java to klasy implementujące `interfejs java.util.Map` (nie implementują `java.util.Collection`). Mapy reprezentują związki obiektów z kluczami. Mapa jest jednoznacznym odwzorowaniem zbioru kluczy w zbiór wartości. O mapach możemy myśleć jako o takich kolekcjach par: **klucz - wartość**, które zapewniają odnajdywanie wartości związanej z podanym kluczem. Mapy są niezwykle użytecznym narzędziem programistycznym, pozwalają bowiem prosto i efektywnie programować wiele realnych problemów. Istotą zastosowania map jest możliwość łatwego i jednocześnie szybkiego odnajdywania informacji w powiązanych zestawach danych.

Wśród gotowych implementacji map w JCF są dostępne następujące klasy:

- **HashMap** - mapa w której wyszukiwanie elementów realizowane jest z użyciem tablicy mieszającej,
- **HashTree** - mapa w której wyszukiwanie elementów realizowane jest z użyciem drzewa czerwono-czarnego,
- **Hashtable** - mapa oparta na tablicy,
- **LinkedHashMap** - mapa oparta na liście.

Interfejs `java.util.Map` umożliwia dodatkowe operacje na mapie:

- `.put()` - dodaje element do mapy oraz przypisuje mu określony klucz,
- `.get()` - zwraca element przypisany do określonego klucza,
- `.remove()` - usuwa element skojarzony z kluczem z mapy, o ile istnieje,
- `.size()` - zwraca liczbę kluczy w mapie,
- `.entrySet()` - zwraca zbiór elementów zawartych w mapie,
- `.keySet()` - zwraca zbiór kluczy zawartych w mapie.

Przykład 4. Przykład użycia HashMap

```
import java.util.HashMap;

public class HashMapExample {

  public static void main(String[] args) {

    HashMap hMap = new HashMap();

    hMap.put("One", new Integer(1));
    hMap.put("Two", new Integer(2));

    Object obj = hMap.get("One");
    System.out.println(obj);
  }
}
```

Iteratory

Iterator jest obiektem klasy implementującej interfejs *Iterator* i służy do przeglądania elementów kolekcji oraz ewentualne usuwania ich przy przeglądaniu. Klasy iteratorów są definiowane w klasach kolekcyjnych jako klasy wewnętrzne, implementujące interfejs `Iterator<T>`. Implementacja metody `iterator()` z interfejsu Collection zwraca obiekt takiej klasy. Dzięki temu od każdej kolekcji możemy uzyskać iterator za pomocą odwołania:

```
Iterator<T> iter = c.iterator();
```

gdzie: c - dowolna klasa implementująca interfejs Collection, T - typ elementów kolekcji.

Dla tych kolekcji, w których elementy nie zajmują ściśle określonych pozycji iteratory są jedynym sposobem na "poruszanie się" po kolekcji. Co więcej, dla kolekcji listowych - niezależnie od implementacji - iteratory są efektywnym narzędziem iterowania, czego nie da się powiedzieć we wszystkich przypadkach o pętlach iteracyjnych pobierających elementy z pozycji wyznaczanych przez podane indeksy.

[Iterator] | *IteratorJava01.gif*

Rysunek 3: Iterator.

Interfejs `java.util.Iterator` umożliwia operacje na kolekcjach:

- `.hasNext()` - zwraca **true**, gdy iteracja posiada więcej elementów,
- `.next()` - zwraca kolejny element iteracji,
- `.remove()` - usuwa z kolekcji ostatni element zwrócony przez iterator.

W trakcie iteracji za pomocą **iteratora** nie wolno modyfikować kolekcji innymi sposobami niż użycie metody `remove()` na rzecz iteratora. Wyniki takich modyfikacji są nieprzewidywalne.

Szczególnie przy stosowaniu **iteratora listowego** należy pamiętać o tym, że iterator *"zajmuje pozycję"* pomiędzy elementami.

Przykład 5. Przykład użycia Iteratora

```
import java.util.Iterator;
import java.util.ArrayList;

public class IteratorExample {

  public static void main(String[] args) {

    ArrayList aList = new ArrayList();

    aList.add("1");
    aList.add("2");
    aList.add("3");
    aList.add("4");
    aList.add("5");

    Iterator itr = aList.iterator();

    while(itr.hasNext())
      System.out.println(itr.next());

  }
}
```

Operacje na tablicach

Przykład 6. Tablica obiektów

```
class A { }

A[] a;
A[] b = new A[5];
A[] c = { new A(), new A(), new A() };
```

Klasa System:

```
java.lang.Object
\--java.lang.System
```

umożliwia operacje na tablicach:

- `.arraycopy()` - kopiowanie (tylko wskaźniki do obiektów).

Klasa Arrays:

```
java.lang.Object
\--java.util.Arrays
```

umożliwia operacje na tablicach:

- `.fill()` - wypełnianie,
- `.equals()` - porównywanie,
- `.sort()` - sortowanie,
- `.binarySearch()` - przeszukiwanie.

Bibliografia

- **Bruce Eckels**, *"Thinking in Java. Edycja polska. Wydanie IV"*, wydawnictwo Helion.
- **Cay S. Horstmann, Gary Cornell**, *"Java. Podstawy. Wydanie IX"*, wydawnictwo Helion.
- **Cay S. Horstmann, Gary Cornell**, *"Java. Techniki zaawansowane. Wydanie IX"*, wydawnictwo Helion.
- **Krzysztof Barteczko**, *"Podstawy programowania w języku Java, PJWSTK"*, http://edu.pjwstk.edu.pl/wyklady/ppj/scb/
- **Konrad Kurczyna**, *"Laboratorium Java"*, Politechnika Świętokrzyska w Kielcach.
- **Mariusz Lipiński**, *"Nauka Javy"*, http://www.naukajavy.pl/