Java Collections Framework

Co to jest Java Collections Framework

- JCF – Zunifikowana architektura do reprezentacji i manipulacji kolekcjami danych.

Składa się z:

- Interfejsów – Definuje abstrakcyjne typy możliwych manipulacji na danych
- Implementacji – implementacji różnych struktur danych
- Algorytmów – algorytmów do operacji na da nych

Żródło: http://docs.oracle.com/javase/tutorial/collections/intro/index.html

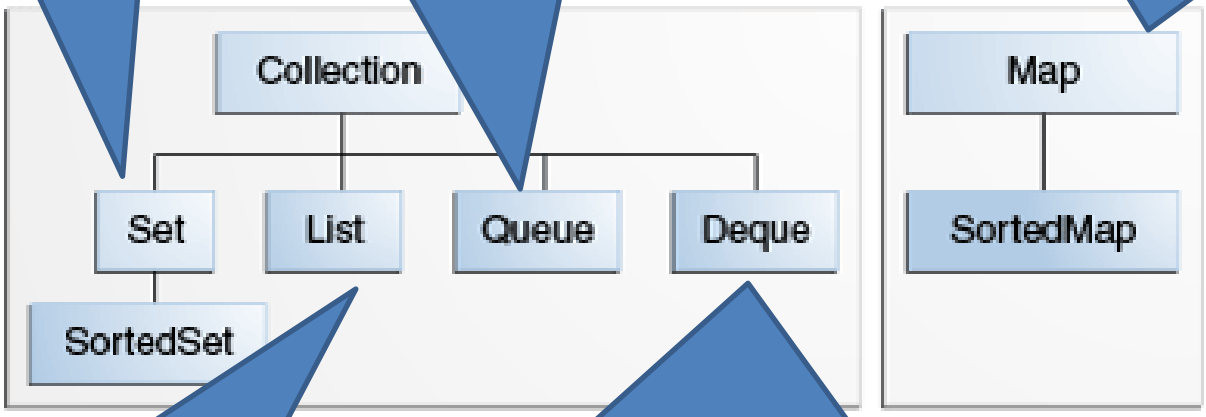
Interfejsy

public interface Collection<E>

Zbiór unikatowych wartości

Kolejka typu FIFO, Gwarantuje kolejność elementów zgodną z kolejnością ich wkładania. Metody: remove i poll zawsze działają na „głowie” kolejki

Słownik mapujący klucz na wartość



Dynamiczna tablica danych

Kolejka typu FIFO, lub LIFO, elementy wkładane do takiej kolejki mogą być układane na obydwu końcach (głowie i ogonie)

Źródło: http://docs.oracle.com/javase/tutorial/collections/intro/index.html

public interface Collection<E>

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Rozmiar – liczba elementów w kolekcji

Czy kolekcja jest pusta

Czy kolekcja zawiera element

Dodanie elementu

Usunięcie elementu

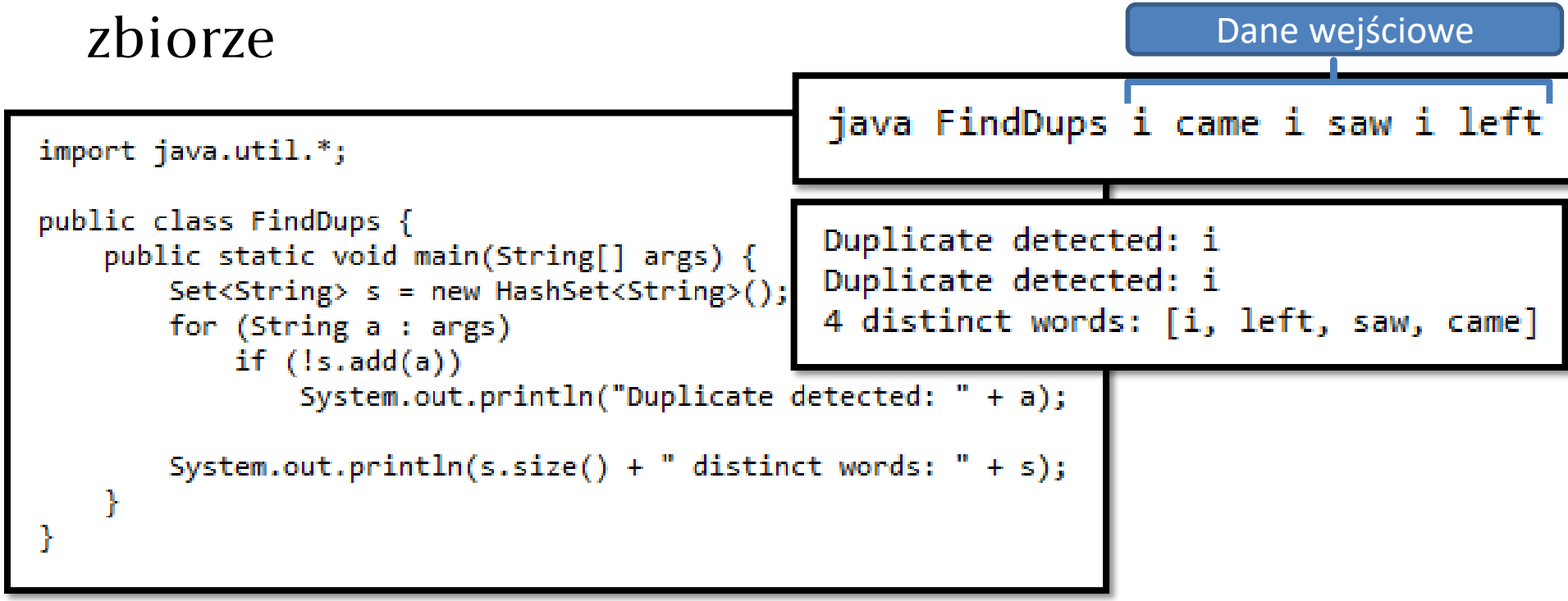
Pobranie iteratora po elementach kolekcji np. petla ForEach

Operacje wsadowe w celu przyspieszenia działania

Operacje tablicowe

SET

- Podczas dodawania elementu automatycznie sprawdza czy dany element nie występuje już w zbiorze



- Jeśli elementu już był to add zwraca *false* (dodanie do zbioru nie udało się)

LIST

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    // optional
    E set(int index, E element);
    // optional
    boolean add(E element);
    // optional
    void add(int index, E element);
    // optional
    E remove(int index);
    // optional
    boolean addAll(int index, Collection<? extends E> c);

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

Odczytanie elementu spod indeksu „i”

Ustawienie elementu pod indeksem „i”

Dodanie elementu w miejsce „i” (rozsuwa listę)

Usunięcie „i” tego elementu

Odczytanie indeksu danego elementu listy

Pobranie iteratora.
UWAGA: Lista ma rozszerzony iterator

Zwraca listę będącą widokiem na oryginalną listę

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove(); //optional
    void set(E e); //optional
    void add(E e); //optional
}
```

Iterator dwukierunkowy

List a algorytmy

- sort — Sortowanie listy
- shuffle — losowe układanie elementów listy
- reverse — odwrócenie listy
- replaceAll — Zamiana wszystkich wystąpień jednej wartości na inną
- fill — Zastąpienie wszystkich elementów list przez specjalną wartość
- copy — kopiowanie list
- binarySearch — Poszukiwanie określonego elementu na liście z wykorzystaniem algorytmu binarySearch – uwaga lista musi być posortowana

Queue

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

Pobiera ale nie usuwa elementu z kolejki

Dodaje element do klejki, zwraca true jeśli dopdanie zakończone powodzeniem

Pobiera ale nie usuwa elementu z kolejki. Jeśli kolejka jest pusta zwraca null

Pobiera ale i usuwa element z kolejki. Jeśli kolejka jest pusta zwraca null

Pobiera ale i usuwa elementu z kolejki.

Zwraca wyjątek: IllegalStateException jeśli przekroczono dozwolony rozmiar kolejki

- Uwaga: Niektóre metody zwracają wyjątek (jak kolejka jest pusta, lub gdy przekroczono dozwolony rozmiar)

Queue In ce Structure

Type of Operation	Throws ception	Returns special value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Deque

- Kolejka dwukierunkowa

Deque Methods

Type of Operation	First Element (Beginning of the `Deque` instance)	Last Element (End of the `Deque` instance)
Insert	`addFirst(e)` `offerFirst(e)`	`addLast(e)` `offerLast(e)`
Remove	`removeFirst()` `pollFirst()`	`removeLast()` `pollLast()`
Examine	`getFirst()` `peekFirst()`	`getLast()` `peekLast()`

Map

```
public interface Map<K,V> {

    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Tablica asocjacyjna

Implementacje

General-purpose Implementations

Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Implementacje Set

- HashSet – szybka implementacja, stała złożoność dla większości operacji
- TreeSet – zwykle złożoność logarytmiczna, ale automatyczne sortowanie elementów

Implementacje List

- ArrayList – bardzo szybka implementacja bazująca na tablicach, problem gdy dodajemy duzo elementów, brak narzutu na posiadanie opakowacza
- LinkedList – Do użytku gdy dodawanych jest dużo elementów na początku listy, lub usuwanie elementów za pomocą iteratora

Implementacje Map

- HashMap – bardzo szybka implementacja bazująca na ArrayList, dobra gdy nie zależy nam na kolejności elementów
- TreeMap – wolna bo dokonuje autosortowanie, ale użyteczna gdy ważna jest kolejność kluczy (np. książka adresowa)

Implementacje Kolejek

- LinkedList – prosta implementacja kolejki

- PriorityQueu – umozliwia tworzenie kolejek priorytetowych – bazuje na komparatorze

Komparator

- Dwie możliwości budowy komparatorów

Wrzucamy do kolekcji elementy implementujące interfejs Comparable

```
class Ksiazka {
    String tytuł;
    String autor;
    int rokWydania;
    public Ksiazka(String tytuł,
            String autor, int rokWydania) {
        this.tytuł = tytuł;
        this.autor = autor;
        this.rokWydania = rokWydania;
    }
}
```

```
class MyComparator implements Comparator<Ksiazka> {
    @Override
    public int compare(Ksiazka o1, Ksiazka o2) {
        if (o1.rokWydania > o2.rokWydania)
            return 1;
        if (o1.rokWydania < o2.rokWydania)
            return -1;
        return 0;
    }
}
```

Tworzymy klasę komparatora – Implementacja interfejsu Comparator Do kolekcji wrzucamy dowolne elementy

```
class KsiazkaCmp implements Comparable<KsiazkaCmp> {
    String tytuł;
    String autor;
    int rokWydania;

    public KsiazkaCmp(String tytuł,
            String autor, int rokWydania) {
        this.tytuł = tytuł;
        this.autor = autor;
        this.rokWydania = rokWydania;
    }

    @Override
    public int compareTo(KsiazkaCmp o) {
        if (this.rokWydania > o.rokWydania) {
            return 1;
        }
        if (this.rokWydania < o.rokWydania) {
            return -1;
        }
        return 0;
    }
}
```

Komparator przykłady

Wrzucamy do kolekcji elementy implementujące interfejs Comparable

```
public class TestCompI {

    public static void main(String[] args) {
        Queue<KsiazkaCmp> q = new PriorityQueue<>();
        q.add(new KsiazkaCmp("Jaś i Małgosia", "Bracia Grimm", 2004));
        q.add(new KsiazkaCmp("Calineczka", "H.C. Andersen", 2006));
        q.add(new KsiazkaCmp("Kajko i Kokosz", "J. Christa", 1970));
        while (!q.isEmpty()) {
            KsiazkaCmp k = q.poll();
            System.out.println(k.rokWydania + " '" + k.tytuł + "' " + k.autor);
        }
```

```
run:
1970 'Kajko i Kokosz' J. Christa
2004 'Jaś i Małgosia' Bracia Grimm
2006 'Calineczka' H.C. Andersen
BUILD SUCCESSFUL (total time: 0 seconds)
```

Tworzymy klasę komparatora – Implementacja interfejsu Comparator Do kolekcji wrzucamy dowolne elementy

```
import java.util.PriorityQueue;
import java.util.Queue;

public class TestComp {
    public static void main(String[] args) {
        Queue<Ksiazka> q = new PriorityQueue<>(10, new MyComparator());
        q.add(new Ksiazka("Jaś i Małgosia", "Bracia Grimm", 2004));
        q.add(new Ksiazka("Calineczka", "H.C. Andersen", 2006));
        q.add(new Ksiazka("Kajko i Kokosz", "J. Christa", 1970));
        while (!q.isEmpty()) {
            Ksiazka k = q.poll();
            System.out.println(k.rokWydania + " '" + k.tytuł + "' " + k.autor);
        }
    }
}
```

Kolejki priorytetowe
Uwaga!!!

```
public class TestCompI {

    public static void main(String[] args) {
        Queue<KsiazkaCmp> q = new PriorityQueue<>();
        q.add(new KsiazkaCmp("Jaś i Małgosia", "Bracia Grimm", 2004));
        q.add(new KsiazkaCmp("Calineczka", "H.C. Andersen", 2006));
        q.add(new KsiazkaCmp("Kajko i Kokosz", "J. Christa", 1970));
        while (!q.isEmpty()) {
            KsiazkaCmp k = q.poll();
            System.out.println(k.rokWydania + " '" + k.tytuł + "' " + k.autor);
        }
    }
}
```

```
run:
1970 'Kajko i Kokosz' J. Christa
2004 'Jaś i Małgosia' Bracia Grimm
2006 'Calineczka' H.C. Andersen
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
public class TestCompI {

    public static void main(String[] args) {
        Queue<KsiazkaCmp> q = new PriorityQueue<>();
        q.add(new KsiazkaCmp("Jaś i Małgosia", "Bracia Grimm", 2004));
        q.add(new KsiazkaCmp("Calineczka", "H.C. Andersen", 2006));
        q.add(new KsiazkaCmp("Kajko i Kokosz", "J. Christa", 1970));
        for (KsiazkaCmp k : q) {
            System.out.println(k.rokWydania + " '" + k.tytuł + "' " + k.autor);
        }
    }
}
```

?

Kolejki priorytetowe Uwaga!!!

```java
public class TestCompI {

    public static void main(String[] args) {
        Queue<KsiazkaCmp> q = new PriorityQueue<>();
        q.add(new KsiazkaCmp("Jaś i Małgosia", "Bracia Grimm", 2004));
        q.add(new KsiazkaCmp("Calineczka", "H.C. Andersen", 2006));
        q.add(new KsiazkaCmp("Kajko i Kokosz", "J. Christa", 1970));
        while (!q.isEmpty()) {
            KsiazkaCmp k = q.poll();
            System.out.println(k.rokWydania + " '" + k.tytuł + "' " + k.autor);
        }
    }
}
```

```
run:
1970 'Kajko i Kokosz' J. Christa
2004 'Jaś i Małgosia' Bracia Grimm
2006 'Calineczka' H.C. Andersen
BUILD SUCCESSFUL (total time: 0 seconds)
```

Kolejka priorytetowa nie gwarantuje poprawnego poprawnej kolejności w iteratorze.
Poprawna kolejność jedynie przy odczytywaniu z głowy lub ogona !!!!!

```java
public class TestCompI {

    public static void main(String[] args) {
        Queue<KsiazkaCmp> q = new PriorityQueue<>();
        q.add(new KsiazkaCmp("Jaś i Małgosia", "Bracia Grimm", 2004));
        q.add(new KsiazkaCmp("Calineczka", "H.C. Andersen", 2006));
        q.add(new KsiazkaCmp("Kajko i Kokosz", "J. Christa", 1970));
        for (KsiazkaCmp k : q) {
            System.out.println(k.rokWydania + " '" + k.tytuł + "' " + k.autor);
        }
    }
}
```

```
run:
1970 'Kajko i Kokosz' J. Christa
2006 'Calineczka' H.C. Andersen
2004 'Jaś i Małgosia' Bracia Grimm
BUILD SUCCESSFUL (total time: 0 seconds)
```

Inne elementy kolekcji

- Klasa java.util.Collections -> zbiór użytecznych metod statycznych do działania na kolekcjach

Metoda	opis
binarySearch	Metoda szuka czy na liście znajduje się określony element i zwraca jego indeks (wymagane Comparator lub Comparable)
disjoint	Sprawdza czy dwie listy są rozłączne, tzn czy elementy na jednej liście nie występują na drugiej, zwraca boolean
frequency	Liczy częstość występowania danego elementu na liście, zwraca częstość tego elementu
min / max	Zwraca najmniejszy / największy element na liście wg. komparatora
reverse	Odwraca porządek elementów na liście
shuffle	Układa elementy listy w sposób losowy
sort	Sortuje elementy listy wg. komparatora
Synchronized(…) np. List	Tworzy wersję synchronizowana określonej kolekcji – czyli ThreadSave
Unmodifiable(…) np. List	Tworzy kolekcję niemodyfikowalną – dużo szybsze w szczególności dla wielowątkowości – brak konieczności synchronizacji – brak sekcji krytycznej

Porównanie

Collection class	Thread-safe alternative	Your data				Operations on your collections						
		Individual elements	Key-value pairs	Duplicate element support	Primitive support	Order of iteration			Performant 'contains' check	Random access		
						FIFO	Sorted	LIFO		By key	By value	By index
HashMap	ConcurrentHashMap	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗
HashBiMap (Guava)	Maps.synchronizedBiMap (new HashBiMap())	✗	✓	✗	✗	✗	✗	✗	✓	✓	✓	✗
ArrayListMultimap (Guava)	Maps.synchronizedMultiMap (new ArrayListMultimap())	✗	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗
LinkedHashMap	Collections.synchronizedMap (new LinkedHashMap())	✗	✓	✗	✗	✓	✗	✗	✓	✓	✗	✗
TreeMap	ConcurrentSkipListMap	✗	✓	✗	✗	✗	✓	✗	✓ *	✓ *	✗	✗
Int2IntMap (Fastutil)		✗	✓	✗	✓	✗	✗	✗	✓	✓	✗	✓
ArrayList	CopyOnWriteArrayList	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗	✓
HashSet	ConcurrentHashMap<Key, Key>	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗
IntArrayList (Fastutil)		✓	✗	✓	✓	✓	✗	✓	✗	✗	✗	✓
PriorityQueue	PriorityBlockingQueue	✓	✗	✓	✗	✗	✓ **	✗	✗	✗	✗	✗
ArrayDeque	ArrayBlockingQueue	✓	✗	✓	✗	✓ **	✗	✓ **	✗	✗	✗	✗

Collection class	Random access by index / key	Search / Contains	Insert
ArrayList	O(1)	O(n)	O(n)
HashSet	O(1)	O(1)	O(1)
HashMap	O(1)	O(1)	O(1)
TreeMap	O(log(n))	O(log(n))	O(log(n))

Remember, not all operations are equally fast. Here's a reminder of how to treat the Big-O complexity notation:

O(1) - constant time, really fast, doesn't depend on the size of your collection

O(log(n)) - pretty fast, your collection size has to be extreme to notice a performance impact

O(n) - linear to your collection size: the larger your collection is, the slower your operations will be

Źródło: http://zeroturnaround.com/wp-content/uploads/2016/04/Java-Collections-cheat-sheet.png