Poo Laboratoare[1]

Contents

[1] http://www.google.ro/

Laborator7

Programare Orientata pe Obiecte: Laborator 7

1 Colecții de obiecte în Java

O colecție este un grup de date manipulate ca un obiect de sine stătător. Colecțiile sunt definite printr-un set de interfețe și au suportul într-un set de clase care implementează aceste interfețe.

1.1 Interfața Iterator

- permite selectarea fiecărui element din colecție

```
Interface Iterator {
       boolean hasNext();
       Object next();
       void remove();
}
```

1.2 Interfața Collection

```
public interface Collection {
       int size();
       boolean isEmpty();
       boolean contains(Object element);
       boolean add(Object element); // Optional
       boolean remove(Object element); // Optional
       Iterator iterator();
       boolean containsAll(Collection c);
       boolean addAll(Collection c); // Optional
       boolean removeAll(Collection c); // Optional
       boolean retainAll(Collection c); // Optional
       void clear(); // Optional
       Object[] toArray();
       Object[] toArray(Object a[]);
}
```

1.3 Interfața Set

- corespunde definiției matematice a unei mulțimi (nu sunt acceptate elemente duplicate)
- în comparație cu interfața *Collection*:
  - interfața este identică
  - fiecare constructor trebuie să creeze o colecție fără duplicate
  - metoda *add()* nu poate adăuga un element care se află deja în mulțime

1.3.1 Clasele HashSet și TreeSet

- implementează interfața *Set*
- clasa *HashSet*:

  - implementată folosind un tabel de dispersie
  - elementele nu sunt ordonate
  - metodele *add()*, *remove()*, *contains()* au complexitate constantă în timp, *O(c)*
- clasa *TreeSet*:
  - implementată folosind un arbore
  - garantează ordonarea elementelor
  - metodele *add()*, *remove()*, *contains()* au complexitate logaritmică în timp, *O(log n)*

1.4 Interfața List

- corespunde unei grupări cu indexare de elemente
- extensii față de interfața *Collection*:
  - accesează elementele prin indecși, la fel ca un vector
  - permite căutarea bazată pe index a elementelor
  - deține un iterator specializat, numit *ListIterator*
  - permite extragerea de subliste
  - metoda *add()* adaugă la sfârșitul listei
  - metoda *remove()* șterge de la începutul listei

```
public interface List extends Collection {
       Object get(int index);
       Object set(int index, Object element); // Optional
       void add(int index, Object element); // Optional
       Object remove(int index); // Optional
       abstract boolean addAll(int index, Collection c);
       int indexOf(Object o);
       int lastIndexOf(Object o);
       ListIterator listIterator();
       ListIterator listIterator(int index);
       List subList(int from, int to);
}
```

1.4.1 Clasele ArrayList și LinkedList

- implementează interfața *List*
- clasa *ArrayList*:
  - implementată folosind un vector
  - elementele pot fi accesate direct prin metodele *get()* și *set()*
- clasa *LinkedList*:
  - implementată folosind o listă dublu înlănțuită
  - metodele *add()* și *remove()* au o performanță mai bună în comparație cu *ArrayList*
  - metodele *get()* și *set()* au o performanța mai slabă în comparație cu *ArrayList*

1.4.2 Interfața ListIterator

```
public interface ListIterator extends Iterator {
       boolean hasNext();
       Object next();
       boolean hasPrevious();
       Object previous();
       int nextIndex();
       int previousIndex();
       void remove(); // Optional
       void set(Object o); // Optional
       void add(Object o); // Optional
}
```

1.5 Interfața Map

- corespunde unei grupări de asocieri cheie-valoare
- poate fi referită ca *dicționar* sau *tablou asociativ*

```
public interface Map {
       Object put(Object key, Object value);
       Object get(Object key);
       Object remove(Object key);
       boolean containsKey(Object key);
       boolean containsValue(Object value);
       int size();
       boolean isEmpty();
       void putAll(Map t);
       void clear();
       public Set keySet();
       public Collection values();
       public Set entrySet();
       public interface Entry {
              Object getKey();
              Object getValue();
              Object setValue(Object value);
       }
}
```

1.5.1 Clasele HashMap și TreeMap

- implementează interfața *Map*
- clasa *HashMap*:
  - implementare bazată pe un tabel de dispersie
  - perechile cheie-valoare nu sunt ordonate
- clasa *TreeMap*:
  - implementare bazată pe arbori roșu-negru
  - perechile cheie-valoare sunt ordonate pe baza cheii

2 Probleme de laborator

2.1 Problema 1

(2 puncte) Să se scrie un program pentru afișarea cuvintelor distincte dintr-un fișier text folosind clasa *TreeSet* (precum și clasele *RandomAccessFile* și *StringTokenizer* sau metoda *split()* din clasa *String*). Fișierul va conține și cuvinte repetate (identice). Elementele mulțimii se vor afișa în ordine crescătoare și în ordine descrescătoare, fără a apela o funcție de sortare, folosind două obiecte *TreeSet* construite diferit (cu și fără argument de tip *Comparator*).

2.2 Problema 2

(2 puncte) Să se definească o clasă *ArraySet* pentru o mulțime neordonată de obiecte realizată ca vector (extinde *ArrayList*), care să poată înlocui o clasă *TreeSet* sau *HashSet* (adică implementează interfața Set). Se vor redefini cele două metode de adăugare a unui obiect:

- boolean add (Object obj)
- void add (int i , Object obj)

Să se folosească clasa *ArraySet* în locul clasei *TreeSet* din programul anterior.

2.3 Problema 3

(2 puncte) Să se definească o clasă *LinkedSet* pentru o mulțime realizată ca listă înlănțuită de elemente distincte. Clasa va implementa interfața *Set* și va extinde clasa *LinkedList*. Să se scrie un program pentru crearea unei liste de obiecte *Integer* și afișarea ei cu și fără folosirea unui iterator.

2.4 Problema 4

(2 puncte) Să se definească o clasă *LinkedSet* pentru o mulțime realizată ca listă simplu înlănțuită, ca o clasă derivată din *AbstractSet* (se va defini și o clasă *Node*). Se vor implementa metodele *add*, *size*, *iterator*, *toString*. Adăugarea de elemente se va face numai la sfârșitul listei.

2.5 Problema 5

(2 puncte) Să se definească o clasă *LinkList* pentru o listă simplu înlănțuită de variabile *Object*, ca o clasă derivată din *AbstractList* (se va defini și o clasa Node). Se vor defini următoarele metode abstracte:

- int size()
- boolean add(Object)
- Object get (int)
- Object set (int, Object)

Să se scrie un program pentru crearea, ordonarea (cu metoda *Collections.sort()*) și afișarea (cu iterator) a unei liste de șiruri, ca obiect *LinkList*.

2.6 Problema 6 (bonus)

(1 punct) Să se definească o clasă *SLinkList* pentru o listă simplu înlănțuită ordonată de variabile *Object*, ca o clasă derivată din *AbstractList*. Față de clasa anterioară se va adăuaga clasei o variabilă *Comparator* și un constructor cu argument *Comparator*. Se vor redefini metodele:

- boolean add(Object) // adăugare la lista ordonată

- int indexOf(Object) // căutare în lista ordonată
- Object set(int, Object) // modificare și reordonare listă

2.7 Problema 7 (bonus)

(1 punct) Să se definească o clasă *SArraySet* pentru o mulțime ordonată de obiecte și care implementează interfața *SortedSet* (poate extinde *ArrayList*). Metodele impuse de interfața *SortedSet* sunt:

- Comparator comparator() // comparator folosit (*null* pentru comparația naturală)
- Object first() // primul obiect din mulțime
- Object last() // ultimul element din mulțime
- SortedSet subset(Object from, Object to) // o submulțime ordonată
- SortedSet headSet(Object to) // o submulțime cu primele obiecte
- SortedSet tailSet (Object from) // o submulțime cu ultimele obiecte

Să se folosească un obiect *SArraySet* în programul pentru afișarea cuvintelor distincte dintr-un text în ordine crescătoare și descrescătoare. Se vor defini (cel puțin) doi constructori: fără argumente și cu argument de tip *Comparator*.