Le Collezioni (Java Collections)

G. Grossi

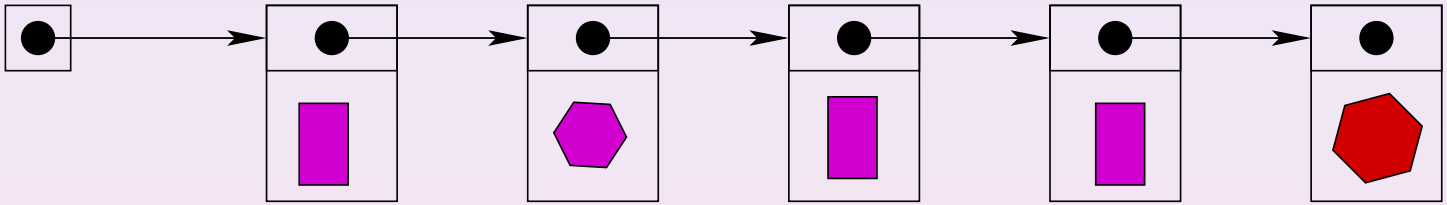
20 dicembre 2006

Indice

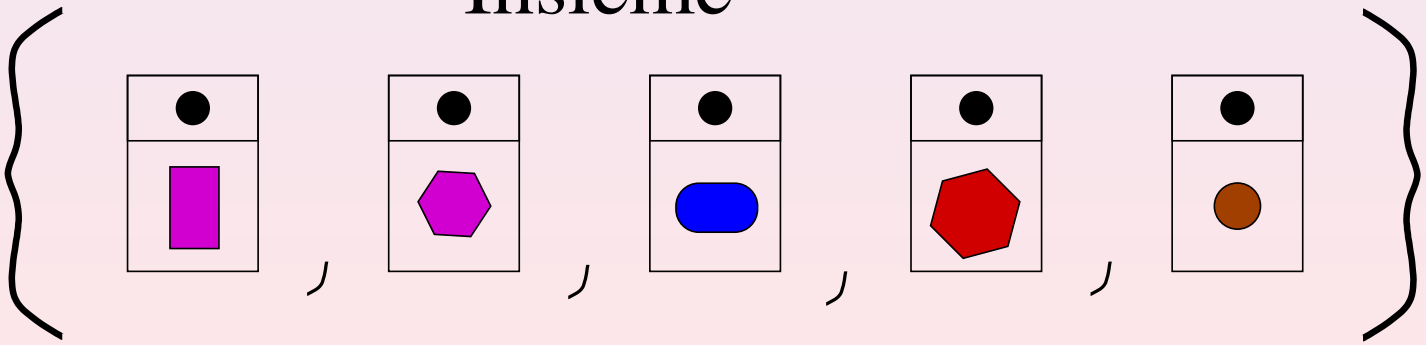
Indice

Liste e insiemi

Lista (sequenza)



Insieme



Il framework Collection

Le collection di Java 2 rappresentano un'architettura unificata preposta all'uso e al trattamento di gruppi di oggetti.
Consistono di:

- Interfacce: tipi di dati astratti che rappresentano collezioni, come `List`, `Queue`, `Set` e `Map`

Il framework Collection

Le collection di Java 2 rappresentano un'architettura unificata preposta all'uso e al trattamento di gruppi di oggetti.
Consistono di:

- Interfacce: tipi di dati astratti che rappresentano collezioni, come `List`, `Queue`, `Set` e `Map`
- Implementazioni astratte: parziali implemetazioni di interfacce facilmente riadattabili

Il framework Collection

Le collection di Java 2 rappresentano un'architettura unificata preposta all'uso e al trattamento di gruppi di oggetti.
Consistono di:

- Interfacce: tipi di dati astratti che rappresentano collezioni, come `List`, `Queue`, `Set` e `Map`
- Implementazioni astratte: parziali implemetazioni di interfacce facilmente riadattabili
- Implementazioni concrete: classi basilari che implementano le interfacce fondamentali

Il framework Collection

Le collection di Java 2 rappresentano un'architettura unificata preposta all'uso e al trattamento di gruppi di oggetti.
Consistono di:

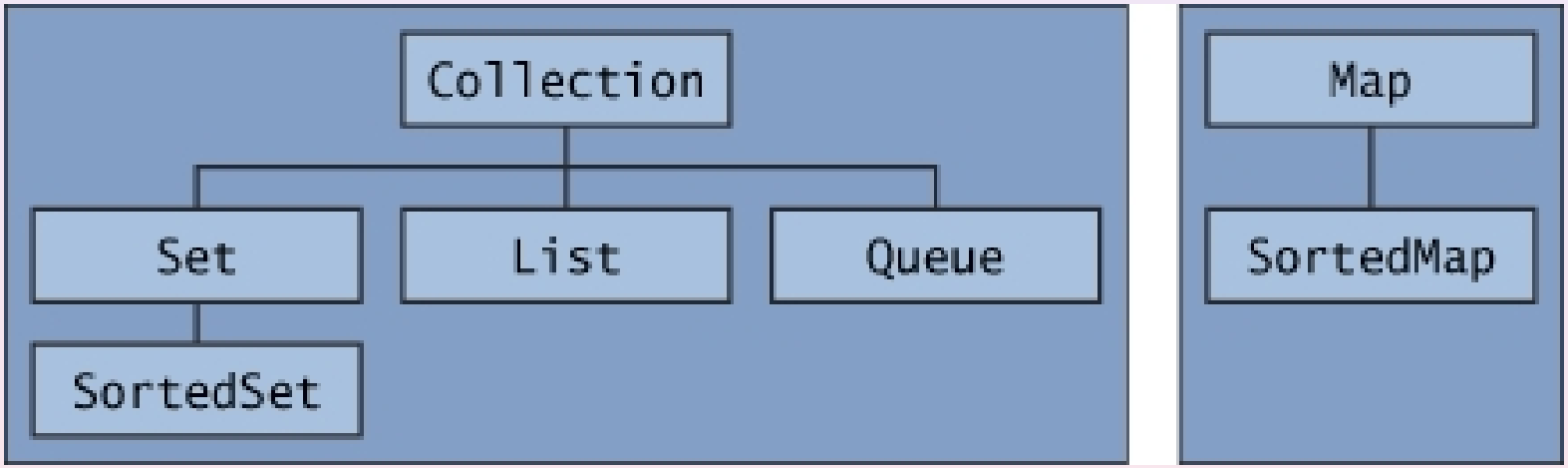
- Interfacce: tipi di dati astratti che rappresentano collezioni, come `List`, `Queue`, `Set` e `Map`
- Implementazioni astratte: parziali implemetazioni di interfacce facilmente riadattabili
- Implementazioni concrete: classi basilari che implementano le interfacce fondamentali
- Algoritmi: implementazioni di funzioni basilari come ordinamento e ricerca, applicabili a tutte le collezioni

Il framework Collection

Le collection di Java 2 rappresentano un'architettura unificata preposta all'uso e al trattamento di gruppi di oggetti.
Consistono di:

- Interfacce: tipi di dati astratti che rappresentano collezioni, come `List`, `Queue`, `Set` e `Map`
- Implementazioni astratte: parziali implemetazioni di interfacce facilmente riadattabili
- Implementazioni concrete: classi basilari che implementano le interfacce fondamentali
- Algoritmi: implementazioni di funzioni basilari come ordinamento e ricerca, applicabili a tutte le collezioni
- Utilità: funzioni basilari applicate ad array di primitivi e riferimento

La gerarchia delle interfacce



L'interfaccia Collection

```
public interface Collection<E> extends Iterable<E> {
    // Operationi base
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);           // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();
    // Operationi sull'insieme
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); // Optional
    boolean removeAll(Collection<?> c);        // Optional
    boolean retainAll(Collection<?> c);        // Optional
    void clear(); // Optional
    // Operationi su array
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Contratto del metodo `add(E o)` - opzionale

- ... Returns `true` if this collection changed as a result of the call. (Returns `false` if this collection does not permit duplicates and already contains the specified element.)
- ... limitations on what elements may be added to this collection. In particular, some collections will refuse to add `null` elements, and others will impose restrictions on the type of elements that may be added. Collection classes should clearly specify in their documentation any restrictions on what elements may be added.
- If a collection refuses to add a particular element for any reason other than that it already contains the element, it must throw an exception (rather than returning false)...

Le interfacce di Collection

`Collection` è la radice della gerarchia delle collection. Rappresenta gruppi di oggetti (elementi) che possono essere duplicati oppure no. Alcune possono essere ordinate altre no. Esistono implementazioni concrete di sottointerfacce quali `List` e `Set`

Le interfacce di Collection

`Collection` è la radice della gerarchia delle collection. Rappresenta gruppi di oggetti (elementi) che possono essere duplicati oppure no. Alcune possono essere ordinate altre no. Esistono implementazioni concrete di sottointerfacce quali `List` e `Set`

`List` cattura il concetto di `lista` ordinata (o sequenza): gruppo di elementi posti in un qualche ordine. Implementazioni: `ArrayList`, `LinkedList` e `Vector`

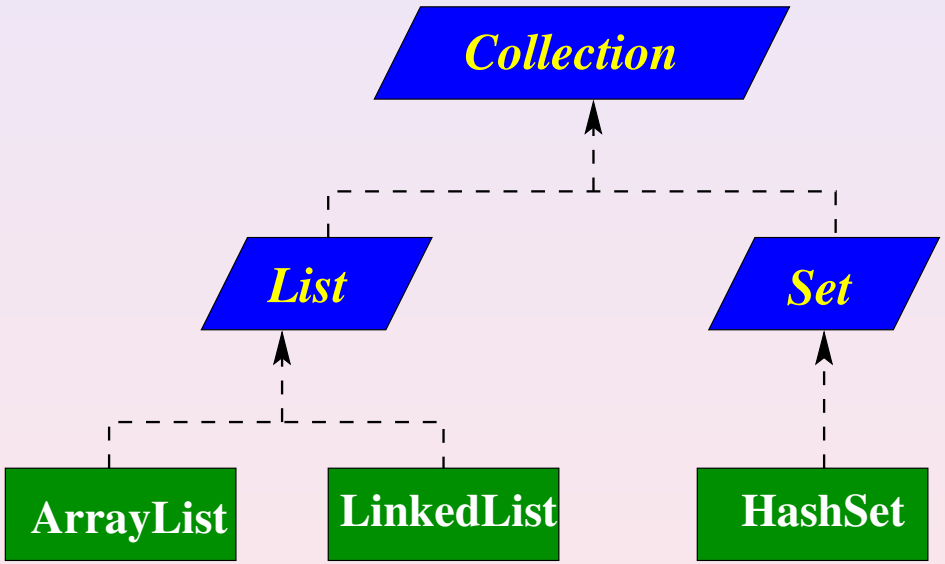
Le interfacce di Collection

`Collection` è la radice della gerarchia delle collection. Rappresenta gruppi di oggetti (elementi) che possono essere duplicati oppure no. Alcune possono essere ordinate altre no. Esistono implementazioni concrete di sottointerfacce quali `List` e `Set`

`List` cattura il concetto di `lista` ordinata (o sequenza): gruppo di elementi posti in un qualche ordine. Implementazioni: `ArrayList`, `LinkedList` e `Vector`

`Set` cattura il concetto di insieme: gruppo di elementi che non contiene duplicati (non contiene **`e1`** e **`e2`** se **`e1.equals(e2)`**. Implementazioni: `HashSet`

Gerarchia (parziale) di Collection (List e Set)



Indice

Esempio

```
class Gatto {
  // costruttore
   public Gatto(int i)
   // restituisce l'identificativo unico
   public int getID()
}


public class Cane {
   // costruttore
   public Cane(int i)
   // restituisce l'identificativo unico
   public int getID()
}
```

Esempio: uso di non-generici

```
import java.util.*;

public class CaniEGatti {
  public static void main(String[] args) {

    /* definizione tipica: uso del
       riferimento piu' astratto */
    List gatti = new ArrayList();
    for(int i = 0; i < 7; i++)
       gatti.add(new Gatto(i));

    /* nessun problema ad aggiungere
       oggetti di tipo diverso */
    for (int i = 0; i < 3; i++)
       gatti.add(new Cane(i));
  }
}
```

Svantaggi

Uno svantaggio tipico è la perdita di informazione sul tipo quando si inseriscono oggetti nel contenitore. Infatti:

- poiché non vi sono restrizioni sul tipo di oggetti inseribili nel contenitore, è possibile inserire senza problemi cani in una collezione di gatti (fonte di potenziali errori)

Svantaggi

Uno svantaggio tipico è la perdita di informazione sul tipo quando si inseriscono oggetti nel contenitore. Infatti:

- poiché non vi sono restrizioni sul tipo di oggetti inseribili nel contenitore, è possibile inserire senza problemi cani in una collezione di gatti (fonte di potenziali errori)
- a causa della perdita del tipo, occorre eseguire una forzatura (`cast`) "corretta", a run time, all'atto dell'estrazione dell'oggetto per non incorrere in un'eccezione

Esempio

```
import java.util.*;

public class CaniEGatti {
  public static void main(String[] args) {

    List gatti = new ArrayList();
    for(int i = 0; i < 7; i++)
      gatti.add(new Gatto(i));

    gatti.add(new Cane(7));

    /* Eccezione rilevata a run time */
    for(int i = 0; i < gatti.size(); i++)
      ((Gatto)gatti.get(i)).id();
  }
}
```

Parametrizzare la lista

Per ovviare ai problemi precedenti è lecito definire una nuova classe lista parametrizzata (nella nuova versione 1.5) ...

```
List<Gatto> gatti = new ArrayList<Gatto>();
List<Cane> cani = new ArrayList<Cane>();

for (int i = 0; i < 7; i++) // aggiungi
  gatti.add(new Gatto(i));

for (int i = 0; i < 3; i++) // aggiungi
  cani.add(new Cane(i));

for (Gatto g : gatti)    // estrai
  out.println(g);
for (Cane c : cani)
  out.println(c);
```

Vantaggi nell'uso di generici

I principali vantaggi dovuti all'introduzione dei tipi generici sono:

- Il compilatore può verificare qulasiasi operazione `add` di oggetti alla collezione
- il tipo dell'oggetto estratto da una collezione è noto, quindi non vi è la necessità di operare un cast a un tipo diverso (fonte di potenziali errori se il cast è fatto su tipi non omologhi)

ArrayList VS LinkedList

ArrayList ottimizzato l'accesso casuale (basato su array), non ottimizzati l'inserimento e l'eliminazione all'interno della lista (grosso modo equivalente a `Vector`)

LinkedList ottimizzato l'accesso sequenziale, l'inserimento e l'eliminazione, indicato per implementare pile (LIFO) e code (FIFO), infatti contiene i metodi: `addFirst()`, `addLast()`, `getFirst()`, `getLast()`, `removeFirst( )`, `removeLast()`

Indice

Scansione: for-each e Iteratori

Il costrutto `for-each`:

```
for (Object o : collection)
  System.out.println(o);
```

L'initerfaccia `Iterator`:

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
}
```

Uso: for-each e Iteratori

Si usa l'iteratore in luogo a for-each quando:

- si deve rimuovere il corrente oggetto (esempio filtraggio)

```
static void filter(Collection c) {
  for (Iterator i = c.iterator(); i.hasNext();) {
    if (!cond(i.next()))
    i.remove();
}
```

- si deve rimpiazzare un elemento nella lista o nell'array durante l'attraversamento

```
ListIterator<E> extends Iterator<E>
```

- occorre iterare su molteplici collezioni simultaneamente

Uso di `Iterator`

Problema

Usare `Iterator` per modificare la classe `CaniEGatti` al fine di eliminare dalla lista `gatti` i gatti con ID dispari e dalla lista `cani` i cani con ID pari

Codifica

```
List<Gatto> gatti = new ArrayList<Gatto>();
List<Cane> cani = new ArrayList<Cane>();

for (int i = 0; i < 7; i++) // aggiungi
  gatti.add(new Gatto(i));

for (int i = 0; i < 3; i++) // aggiungi
  cani.add(new Cane(i));

// iterator
for (Iterator<Gatto> eG = gatti.iterator(); eG.hasNext();){
  if ((eG.next().getID() % 2) == 1)
    eG.remove();
}
for (Iterator<Cane> eC = cani.iterator(); eC.hasNext();) {
  if ((eC.next().getID() % 2) == 0)
    eC.remove();
}
```

Indice

L'interfaccia Set

```
public interface Set<E> extends Collection<E> {
    // Operationi base
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);           // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Operationi su collezioni
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); // Optional
    boolean removeAll(Collection<?> c);        // Optional
    boolean retainAll(Collection<?> c);        // Optional
    void clear();                              // Optional

    // Operationi su array
    Object[] toArray();
    <T> T[] toArray(T[] a);
```

La classe HashSet

Una della classi più utili nella gestione di insieme di oggetti è la classe `HashSet`

```
Set<T> insieme = new HashSet<T>(c);
```

Osservazione: Notare che si usa il tipo interfaccia `Set` per la variabile `insieme` (pratica fortemente raccomadata). Si produce codice molto flessibile. Per esempio, per avere l'elenco ordinato:

```
Set<T> insieme = new TreeSet<T>(c);
```

Il codice hash associato agli oggetti

Funzione hash

Un funzione hash in Java è una funzione che mappa un oggetto in un numero intero (codice hash)

Proprietà:

- oggetti logicamente uguali devono avere lo stesso codice hash
- Oggetti distinti possono avere lo stesso codice hash (collisione), l'importante è che questo fatto sia piuttosto infrequente
- Deve essere una funzione semplice e "veloce" dal calcolare, possibilmente dipendente da campi della classe

Il metodo `hashCode()`

Contratto di `Object.hashCode`

Il metodo `hashCode` deve produrre lo stesso numero anche quando invocato più volte sullo stesso oggetto. Se due oggetti sono uguali in accordo al metodo `equal(Object o)`, allora il metodo `hashCode` invocato su entrambi gli oggetti deve produrre lo stesso numero. Non è richiesto che oggetti diversi producano codici hash distinti

Osservazioni: normalmente questo contratto viene disatteso! Per usare `HashSet` è fondamentale osservarlo!
Regola: si deve sovrascrivere `hashCode()` in tutte le classi ove viene sovrascritto `equal(Object o)`

Esempi di codici hash

```
...
System.out.println("cane".hashCode());
System.out.println("cane".hashCode());
System.out.println("gatto".hashCode());
Quadrato q = new Quadrato(1);
System.out.println(q.hashCode());
q = new Quadrato(1);
System.out.println(q.hashCode());
q = new Quadrato(2);
System.out.println(q.hashCode());
Persona p = new Persona("Paolo","Rossi");
System.out.println(p.hashCode());
p = new Persona("Paolo","Rossi");
System.out.println(p.hashCode());
p = new Persona("Paola","Rossi");
System.out.println(p.hashCode());
...
```

Stampa dei Risultati

Esecuzione del metodo main della classe `HashCodeEx`:

```
HashCode di: "cane" -> 3046037
HashCode di: "cane" -> 3046037
HashCode di: "gatto" -> 98127573
HashCode di: Quadrato(1) -> 19972507  //non definito
HashCode di: Quadrato(1) -> 7754385   //non definito
HashCode di: Quadrato(2) -> 2548785   //non definito
HashCode di: Persona("Paolo","Rossi") -> -1832489941
HashCode di: Persona("Paolo","Rossi") -> -1832489941
HashCode di: Persona("Paola","Rossi") -> -1832490375
```

Uso di `HashSet`

Problema

Usare la classe `HashSet` (`TreeSet`) per trovare le parole duplicate in una stringa passata come argomento sulla linea di comando

```
>java TrovaDuplicati ciao mamma ciao

Duplicati: ciao
2 parole distinte: [mamma, ciao]

Duplicati: ciao
2 parole distinte (ordinate): [ciao, mamma]
```

Esempio

```
import java.util.*;

class TrovaDuplicati {
  public static void main(String args[]) {

    Set<String> s = new HashSet<String>();
    for (String a : args)
       if (!s.add(a))
          System.out.println("Duplicati: " + a);
    System.out.println(s.size() + " parole distinte: " + s);

    s = new TreeSet<String>();
    for (String a : args)
       if (!s.add(a))
          System.out.println("Duplicati: " + a);
    System.out.println(s.size() + " parole distinte (ordinate): " + s)
  }
}
```

Indice

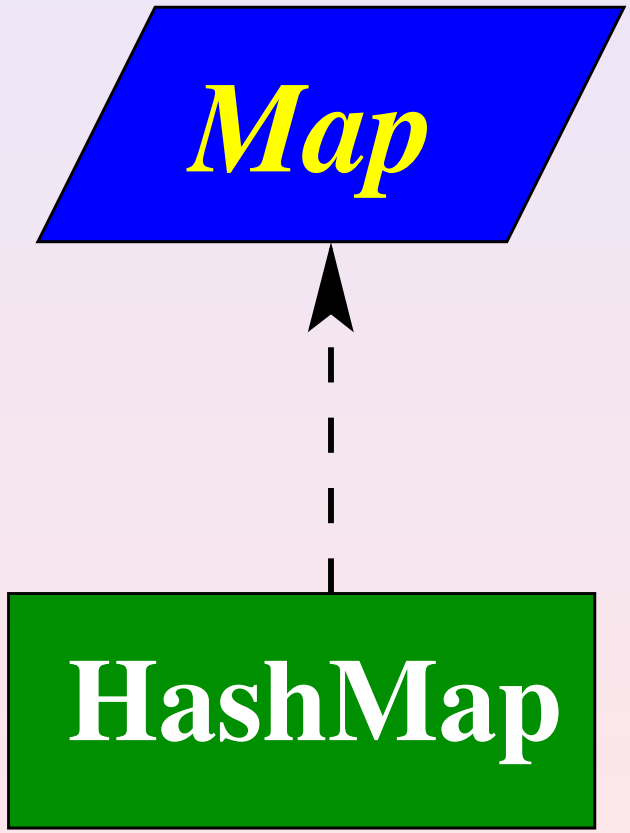
L'interfaccia Mappa

```
public interface Map<K,V> {
    // Operationi base
    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();
    // Operationi su insiemi
    void putAll(Map<? extends K,? extends V> t);
    void clear();

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

Mappe

Map Un gruppo di coppie di oggetti chiave-valore. Una `Map` può restiuire un `Set` di chiavi, una `Collection` di valori o un `Set` delle sue coppie.
Idea: associare oggetti (chiave) a oggetti (valore), invece di associare numeri a oggetti (come `ArrayList`)



Uso di `HashMap`

Problema

Usare la classe `HashMap` (`TreeMap`) per calcolare le occorrenze di ogni singola parola in una stringa passata come argomento sulla linea di comando

```
>java FrequenzaParole ciao mamma ciao

2 parole distinte:
{mamma=1, ciao=2}

2 parole distinte (ordinate):
{ciao=2, mamma=1}
```

Esempio

```
public class FrequenzaParole {
  public static void main(String args[]) {
        Map<String, Integer> m = new HashMap<String, Integer>();
        for (String a : args) {
            Integer freq = m.get(a);
            m.put(a, (freq == null ? 1 : freq + 1));
        }
        System.out.println(m.size() + " parole distinte:");
        System.out.println(m);

        m = new TreeMap<String, Integer>();
        for (String a : args) {
            Integer freq = m.get(a);
            m.put(a, (freq == null ? 1 : freq + 1));
        }
        System.out.println(m.size() + " parole distinte:");
        System.out.println(m);
  }
}
```

Problema

Problema

Scrivere un programma che mostri la pseudo casualità del generatore di numeri casuali implementato nella classe `java.util.Random` utilizzando la classe `Contatore` per contare le occorrenze dei numeri.

Metodo: genero molti (10000) numeri nell'intervallo $[1, 10]$ e mostro la frequenza con cui si manifestano, quindi stampo la coppia (numero, frequenza)

Risultato

```
>java DistUniformeDemo
Frequenze di 10000 numeri (tra 1 e 10) generati a caso:

[1=974, 2=1000, 3=954, 4=1010, 5=993, 6=1049, 7=996,
 8=1000, 9=977, 10=1037]
```

Problema con mappe ordinate per chiave

Problema

Scrivere un programma che gestisca una rubrica telefonica: una chiave (ordinabile), per es. cognome e un valore per es. Persona.
Si consiglia l'uso di `java.util.TreeMap`