Esame di Programmazione II, 6 settembre 2016

Esercizio 1 [8 punti] Un `PhoneBook` è una raccolta di record contenenti nome, cognome, numero di telefono e genere (maschile/femminile) di persone. Si completi la sua implementazione, riempiendo le parti mancanti nella classe che segue:

```
public class PhoneBook extends View {
  ...
  public static class Entry {
    public final String name;      public final String surname;
    public final int phone;        public final boolean sex;
    public final static boolean MALE = false, FEMALE = true;

    private Entry(String name, String surname, int phone, boolean sex) {
      this.name = name;    this.surname = surname;
      this.phone = phone;  this.sex = sex;
    }

    @Override public String toString() {
      return name + " " + surname + ": " + phone + (sex == MALE ? " [male]" : " [female]");
    }
  }

  public void add(String name, String surname, int phone, boolean sex) { ...
    // aggiunge l'entry; se gia' ne esisteva una con lo stesso nome e cognome, la sostituisce
  }

  public void remove(String name, String surname) { ...
    // rimuove l'entry con tale nome e cognome; se non e' presente, lancia una UnknownEntryException
  }

  @Override public Iterator<Entry> iterator() { ...
    // resituisce un iteratore sulle entry di questo PhoneBook
  }
}
```

Esercizio 2 [2 punti] Si implementi l'eccezione `UnknownEntryException`.

Esercizio 3 [10 punti] Una `View` è un iterabile di entry, cioè un oggetto che, se iterato, fornisce un'entry alla volta. La sua implementazione è la seguente. Si noti che si tratta di una classe astratta quindi il suo metodo `iterator()` deve essere implementato nelle sue sottoclassi concrete, come ad esempio l'`AddressBook` visto prima:

```
public abstract class View implements Iterable<Entry> {
  protected View() {}

  @Override public final String toString() {
    String result = "";
    for (Entry entry: this)
      result += entry.toString() + "\n";

    return result;
  }
}
```

Si definisca una sottoclasse concreta `SexView` di `View` che, data un'altra `View` padre, è un iterabile sulle sole entry del padre che abbiano un genere ben preciso, fornito al costruttore. Si completi a tal fine la seguente implementazione:

```
public class SexView extends View {
  private final View parent;  private final boolean sex;
  public SexView(View parent, boolean sex) { this.parent = parent;  this.sex = sex; }
  @Override public Iterator<Entry> iterator() { ... }
}
```

Esercizio 4 [11 punti, solo per chi non ha già fatto il progetto degli anni passati] Si definisca una sottoclasse concreta `SortedView` di `View` che, data un'altra `View` padre, è un iterabile sulle stesse entry del padre, ma nell'ordine specificato da un comparatore fornito al costruttore. Si completi a tal fine la seguente implementazione:

```
public class SortedView extends View {
  private final View parent;  private final Comparator<Entry> comparator;

  public SortedView(View parent, Comparator<Entry> comparator) {
    this.parent = parent; this.comparator = comparator;
  }

  @Override public Iterator<Entry> iterator() { ... }
}
```

Si ricordi che l'interfaccia di libreria `java.util.Comparator` è la seguente:

```
public interface Comparator<T> {
  int compare(T o1, T o2);
}
```

dove il metodo `compare()` ritorna un numero negativo se `o1` viene prima di `o2`, ritorna un numero positivo se `o2` viene prima di `o1` e ritorna 0 altrimenti.

Si ricordi inoltre che una `java.util.List entries` può essere convertita in array con l'istruzione:

```
Entry[] array = entries.toArray(new Entry[entries.size()])
```

e che un `array` può essere ordinato rispetto a un comparatore con l'istruzione:

```
java.util.Arrays.sort(array, comparator)
```

Se tutto è corretto, l'esecuzione del programma:

```
public class Main {
  public static void main(String[] args) {
    PhoneBook book = new PhoneBook();
    book.add("Primo", "Levi", 7569222, MALE);
    book.add("Fausto", "Spoto", 1234567, MALE);
    book.add("Giorgio", "Levi", 1563423, MALE);
    book.add("Catherine", "Bach", 367745, FEMALE);
    book.add("Pietro", "Ferrara", 7865634, MALE);
    book.add("Alberto", "Lovato", 8746728, MALE);
    book.add("Thomas", "Scudiero", 453678, MALE);
    book.add("Pietro", "Ferrara", 333334, MALE); // sostituisce
    book.add("Cybill", "Shepherd", 987567546, FEMALE);
    book.add("Audrey", "Hepburn", 32444, FEMALE);
    book.remove("Fausto", "Spoto");

    View view = book;
    System.out.println("DIRECT VIEW");
    System.out.println(view);

    view = new SexView(view, MALE);
    System.out.println("MALE ONLY VIEW");
    System.out.println(view);

    Comparator<Entry> comparator = new Comparator<Entry>() { // ordina per cognome e poi per nome

      @Override public int compare(Entry address1, Entry address2) {
        int diff = address1.surname.compareTo(address2.surname);
        if (diff != 0)
          return diff;
        else
          return address1.name.compareTo(address2.name);
      }
    };

    view = new SortedView(view, comparator);
    System.out.println("SORTED MALE ONLY VIEW");
    System.out.println(view);
  }
}
```

dovrà stampare:

```
DIRECT VIEW
Primo Levi: 7569222 [male]
Giorgio Levi: 1563423 [male]
Catherine Bach: 367745 [female]
Alberto Lovato: 8746728 [male]
Thomas Scudiero: 453678 [male]
Pietro Ferrara: 333334 [male]
Cybill Shepherd: 987567546 [female]
Audrey Hepburn: 32444 [female]

MALE ONLY VIEW
Primo Levi: 7569222 [male]
Giorgio Levi: 1563423 [male]
Alberto Lovato: 8746728 [male]
Thomas Scudiero: 453678 [male]
Pietro Ferrara: 333334 [male]

SORTED MALE ONLY VIEW
Pietro Ferrara: 333334 [male]
Giorgio Levi: 1563423 [male]
Primo Levi: 7569222 [male]
Alberto Lovato: 8746728 [male]
Thomas Scudiero: 453678 [male]
```

È possibile definire campi, metodi, costruttori e classi aggiuntive, ma solo `private`.