Interfacce Iterator e Iterable

Il problema appena affrontato è così frequente che Java lo ha “standardizzato” prevedendo classi “generiche” Iterator e Iterable

Il termine **Interfaccia** indica una definizione di classe di cui si dichiara la firma dei metodi ma non il relativo sviluppo di codice.

Sarà compito di altre classi **implementare** l'interfaccia sviluppandone tutti i metodi.

Nell'implementazione è obbligatorio implementare tutti i metodi previsti dall'interfaccia, mentre è liberamente possibile prevedere altri metodi

```
interface MiaInterfaccia{
  public void faiQualcosa(); //NOTA BENE: niente codice del metodo
}
class MiaImplementazioneChiaccherona implements MiaInterfaccia {
  public void faiQualcosa(){
    System.out.println(“Io parlo!”);
  }
}
class MiaImplementazioneSilenziosa implements MiaInterfaccia {
  int conta=0;
  public void faiQualcosa(){
    conta++;
  }
  public int quantoHaiFatto(){
    return conta;
  }
}
```

Per essere più precisi le interfacce Iterator e Iterable sono parametrizzate, cioè vanno applicate ad una classe; la scrittura più appropriata è Iterator<E> e Iterable<E> dove al posto di E andrà di volta in volta la classe di riferimento, quasi come se fosse un parametro di un metodo.

Interfaccia Iterator<E>

http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html

Per essere più concreti nell'esempio di squadre e atleti la classe di riferimento è Atleta, per cui la nostra classe iteratore diventa:

```
class SquadraIterator implements Iterator<Atleta> {
  //il codice di implementazione
}
```

i metodi da implementare sono tre di cui due (non a caso) sono hasNext() e next() che abbiamo già visto, il terzo remove() è destinato alla rimozione dell'elemento corrente, ed ecco una possibile implementazione:

```
  /**
   * Classe iteratore sulla squadra
   */
  class SquadraIterator implements Iterator<Atleta> {
```

```
    int indice=0;
    boolean enableRemove=false;

    /**
     * ritorna tru ese il prossimo next() avrà successo
     * @return
     */
    public boolean hasNext(){
      enableRemove=indice<numeroAtleti;
      return enableRemove;
    }

    /**
     * ritorna il prossimo dato
     * @return
     */
    public Atleta next(){
      if (hasNext()){
        Atleta a=squadra[indice];
        indice++;
        return a;
      }
      else {
        throw new NoSuchElementException();
      }
    }

    public void remove(){
      if (enableRemove){
        for (int i=indice; i<numeroAtleti;i++){
          squadra[i]=squadra[i+1];
        }
        squadra[numeroAtleti]=null;
        numeroAtleti--;
        enableRemove=false;
      }
      else {
        throw new IllegalStateException();
      }
    }
  }
```

Interfaccia Iterable<E>

http://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html

Ci si aspetta dunque che ogni classe che implemtenta Iterable abbia un metodo iterator() che restituisce un iteratore sugli elementi interni alla classe stessa,

quindi nel nostro caso diventa:

```
/**
 * Gestione dei dati di una squadra di atleti
 * @author Gianni
 */
public class SquadraDiAtleti implements Iterable<Atleta>{

  // . . .

  /**
   * ritorna un iteratore
   * @return
   */
  public Iterator<Atleta> iterator(){
    return new SquadraIterator();
  }

  /**
   * Classe iteratore sulla squadra
   */
  private class SquadraIterator implements Iterator<Atleta> {

    // . . .

  }
}
```

Si noti che la classe SquadraIterator è diventata privata e che il metodo iterator() dichiara di restituire Iterator<Atleta>.

Dal momento che l'interfaccia di Iterator è nota non ho alcun bisogno di conoscere i dettagli di implementazione per utilizzarla, anzi sabendo che un oggetto è di classe Iterator so esattamente che ho disponibili i tre metodi hasNext(), next() e remove(), mentre per un oggetto di classe SquadraIterator ho bisogno di esaminare il suo elenco di metodi per poterne conoscere le properietà.

Proprietà di un oggetto di classe Iterable

La potenza delle interface sta nel fatto che consentono di generalizzare un problema, di permettere di scrivere codice prima che ancora sia stata pensata una implementazione, di scrivere codice applicabile a tutte le possibili implementazioni.

Vediamo un esempio concreto. Per definizione stessa Iterable si sa che un oggetto di una classe che lo implementaavrà un metodo che restituisce un Iterator, e ancora si è certi che quell'Iterator ha a disposizione il metodo hasNext() e next(), per cui la java virtual machine rende disponibile la possibilità di utilizzare un for applicato ad un oggetto di classe Iterable:

```
for (ElementClass element:IterableObject) {
  // utilizzo element
}
```

Nel nostro esempio vediamo il segmento di codice di scrittura su file di tutti i dati di squadra:

```
        Iterator<Atleta> iteratore= atleti.iterator();
```

```
        while (iteratore.hasNext()){
          Atleta a=iteratore.next();
          scrittore.setPettorale(a.getPettorale());
          scrittore.setCognomeNome(a.getCognomeNome());
          scrittore.setPeso(a.getPeso());
        }
```

la classe degli elementi è Atleta, l'oggetto di classe Iterable è atleti, all'interno del ciclo ci serve un oggetto atleta, quindi possiamo scrivere:

```
        for (Atleta a:atleti){
          scrittore.setPettorale(a.getPettorale());
          scrittore.setCognomeNome(a.getCognomeNome());
          scrittore.setPeso(a.getPeso());
        }
```

questo codice è più semplice da scrivere e rende “invisibile” l'iteratore lasciando evidente quello che è più interessante per l'algoritmo, cioè che partiamo dall'oggetto atleti e lavoriamo su ogni (**foreach**) atleta. Inoltre l'interprete della JVM traduce in modo più efficiente il costrutto così ottenuto.

cc