Il collection framework

Contiene tutte le interfacce e classi che realizzano insiemei ordinati o no di elementi

---

L'interfaccia `Collection`

Dalle API:

> Interface Collection: the root interface in the collection hierarchy. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered.

Idea: mettere in una interfaccia tutte le caratteristiche comuni degli insiemi di oggetti (ordinati o no, con duplicazioni o no).

Metodi di `Collection`

I più importanti:

`add(Object o)`
: aggiungi un oggetto a una collezione

`boolean contains(Object o)`
: vede se un oggetto è contenuto nella collezione

`boolean isEmpty()`
: verifica se la collezione è vuota

`Iterator iterator()`
: fornisce un iteratore per fare una scansione della collezione

`boolean remove(Object o)`
: rimuove un elemento dalla collezione

`int size()`
: numero di elementi della collezione

Nota sul metodo `remove`

Il metodo `remove` ritorna un booleano

Questo booleano dovrebbe valere `True` se la collezione cambia come risultato della `remove` (l'elemento da eliminare era presente)

Contratto e firma

Dei metodi delle interfacce si distinguono:

firma
: l'intestazione del metodo

contratto
: una definizione (a parole!) di cosa si intende che faccia il metodo

Per esempio, il contratto di `remove` è che si tratta di un metodo che rimuove un elemento e che il valore di ritorno indica se la rimozione è stata effettuata.

La firma dice soltanto che è un metodo che ha un `Object` come argomento e un `boolean` come valore di ritorno

In Java si può solo specificare la firma

Il contratto viene dato a parole

Contratto e firma

Se non si rispetta il contratto o la firma:

Non si rispetta	quando per esempio	cosa succede
firma	viene implementato un metodo `remove` che non ha come argomento un `Object` o un valore di ritorno `boolean`	la classe non implementa l'interfaccia, e quindi viene segnalato un errore
contratto	il metodo `remove` ha la firma giusta, ma invece di rimuovere l'elemento lo aggiunge	chi usa la classe non riesce a far funzionare i suoi programmi

Non rispettare la firma è un errore di linguaggio

Non rispettare il contratto è un errore semantico

Costruttori

Le interfacce non possono contenere costruttori (è nel linguaggio)

Il contratto dell'interfaccia `Collection` specifica però che ci deve essere un costruttore vuoto, che crea la collezione senza elementi

---

Esempio di implementazione

Le classi che implementano direttamente `Collection` rappresentano multi-insiemi non ordinati di oggetti.

Esempio di implementazione: usiamo una lista.

```
import java.util.*;

class MultiSet implements Collection {
private LinkedList l;

public MultiSet() {
l=new LinkedList();
}
```

```
public void add(Object o) {
l.add(o);
}

public boolean contains(Object o) {
return l.contains(o);
}

public boolean isEmpty() {
return l.isEmpty();
}

public Iterator iterator() {
return l.iterator();
}

public boolean remove(Object o) {
return l.remove(o);
}

public int size() {
return l.size();
}

// mancano altri metodi
}
```

Implementazione: commenti

Posso aggiungere un metodo `add(int i, Object p)`?

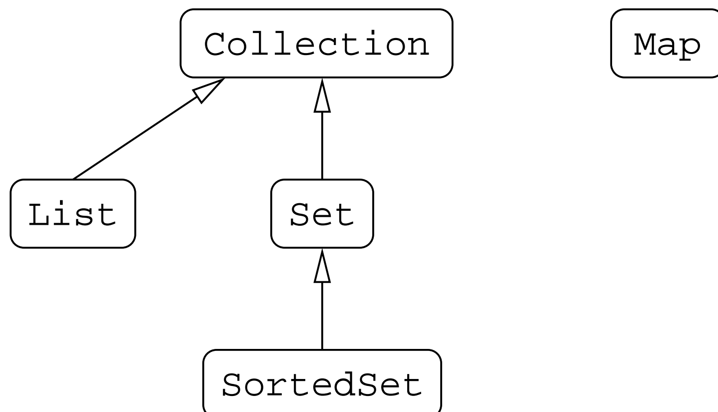
nel linguaggio
: quando si implementa una interfaccia: si devono implementare tutti i metodi, più tutti quelli che voglio

nel contratto
: un `MultiSet` deve rappresentare insiemi non ordinati; quindi, un metodo `add(i, o)` è concettualmente sbagliato

---

Gerarchia delle interfacce

Le interfacce del collection framework sono organizzate cosí:



Il loro significato:

`Collection`
: un multiinsieme non ordinato

`List`
: una sequenza ordinata di elementi

`Set`
: un insieme (non ordinato) di elementi

`SortedSet`
: un insieme ordinato per `compareTo`

`Map`
: un tabella (vediamo tra un momento)

Differenza `List` e `SortedSet`

`List`
: gli elementi sono in una sequenza qualsiasi

`SortedSet`
: gli elementi sono ordinati secondo l'ordinamento specificato da `compareTo`

Attenzione! Nell'interfaccia `SortedSet` non esiste il metodo `get(int index)`

Gli iteratori di `SortedSet` scandiscono l'insieme in ordine crescente

---

Esempio di `SortedSet`

La classe `TreeSet` implementa l'interfaccia `SortedSet`

Esempio: inserisco alcuni interi e poi li stampo:

```
  public static void main(String args[]) {
    SortedSet s=new TreeSet();

    s.add(new Integer(21));
    s.add(new Integer(0));
    s.add(new Integer(4));

    Iterator i=s.iterator();

    while(i.hasNext())
      System.out.println(i.next());
  }
```

Viene stampato:

```
0
4
21
```

Gli iteratori scandiscono l'insieme in ordine

L'ordinamento che is usa è quello dato da `compareTo`

Ordinamento con comparatore

Si puè creare un oggetto `TreeSet` che non usa l'ordinamento dato da `compareTo`

1. si crea una classe che implementa l'interfaccia `Comparable`
2. questa classe deve contenere un metodo (dinamico) `int compare(Object o1, Object o2)`
3. quando si crea l'oggetto `TreeSet`, si specifica un oggetto (qualsiasi) di questa classe

Gli elementi sono nell'ordine dato da `compare` di questa classe, e non più nell'ordine di `compareTo` degli oggetti dell'insieme

---

Esempio di comparatore

```
class Compara implements Comparator {
  public int compare(Object o1, Object o2) {
    int i, j;

    i=((Integer) o1).intValue();
    j=((Integer) o2).intValue();

    if(i<j)
      return 1;
    else if(i==j)
      return 0;
    else
      return -1;
  }
}
```

È l'ordine inverso di confronto fra interi

Il programma di prima, modificato:

```
class Inverso {
  public static void main(String args[]) {
    Compara c=new Compara();
    SortedSet s=new TreeSet(c);

    s.add(new Integer(21));
    s.add(new Integer(0));
    s.add(new Integer(4));

    Iterator i=s.iterator();

    while(i.hasNext())
      System.out.println(i.next());
  }
}
```

Ora viene stampato:

```
21
4
0
```

Le mappe

Servono a implementare "tavole con due colonne"

Esempio: una tavola con un numero di matricola e il nome dello studente:

980123	"Ciccio"
879231	"Aldo"
843098	"Totti"
732108	"Aldo"

Due studenti possono avere lo stesso nome, ma non lo stesso numero di matricola

Gli oggetti mappa sono una generalizzazione

Astrazione

Una tabella è caratterizzata da:

- numero arbitrario di righe
- in ogni riga ci sono due caselle
- la prima casella contiene un identificatore univoco
  (nell' esempio: numero di matricola)

Due caselle della prima riga non possono contenere lo stesso valore

Le due colonne non hanno lo stesso significato:

prima colonna
    un valore che permette di individuare univocamente la riga
seconda colonna
    un valore qualsiasi

`Map` e `HashMap`

`Map`
    interfaccia
`HashMap`
    un oggetto `HashMap` rappresenta una tabella
`TreeMap`
    una implementazione di `Map` che usa algoritmi diversi

Key e value

key
: un oggetto della prima colonna

value
: un oggetto della seconda collonna

Metodi di `Map` e `HashMap`

Metodi fondamentali:

`void put(Object key, Object value)`
: inserisce la riga composta da `key` e `value` nella tabella

`boolean containsKey(Object key)`
: vede se la tabella contiene una riga il cui primo elemento è `key`

`Object get(Object key)`
: se la tabella contiene una riga con `key` in prima colonna, ritorna l'elemento in seconda

---

Creare una tabella

980123	"Ciccio"
879231	"Aldo"
843098	"Totti"
732108	"Aldo"

Questa tabella si può creare cosí:

```
  public static void main(String args[]) {
    HashMap m;

    m=new HashMap();    // tabella vuota

    m.put(new Integer(980123),
          "Ciccio");

    m.put(new Integer(879231),
          "Aldo");

    m.put(new Integer(843098),
          "Totti");

    m.put(new Integer(732108),
          "Aldo");

  }
```

In memoria: la tabella contiene in effetti i *riferimenti* agli oggetti inseriti

Presenza elemento

Per verificare se c'è uno studente con una certa matricola:

```
  public static void main(String args[]) {
    HashMap m;

    // creazione tabella

    if(m.containsKey(new Integer(843040)))
      System.out.println("Matricola "+843040+" esistente");
    else
      System.out.println("Matricola "+843040+" non esistente");

    if(m.containsKey(new Integer(843098)))
      System.out.println("Matricola "+843098+" esistente");
    else
      System.out.println("Matricola "+843098+" non esistente");

  }
}
```

Trovare un elemento

Trovare un elemento con una certa matricola:

```
  public static void main(String args[]) {

    // creazione tabella
    // verifica presenza matricola 843098

    System.out.println("Lo studente con matricola "+843098+
                       " e': "+m.get(new Integer(843098)));
  }
}
```

Generalizzazione

Come key e value, al posto del numero di matricola e del nome, posso usare oggetti qualsiasi

Esempio: numero di matricola e oggetto `Studente`

Realizzazione di `HashSet`

In effetti, gli `HashSet` sono realizzati usando una `HashMap` in cui si usa solo il campo `key`

Gli iteratori

Un interatore è un oggetto che permette la scansione di un insieme

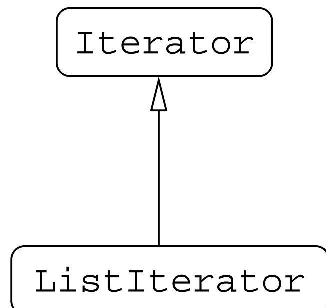
Esiste un metodo della interfaccia `Collection` che restituisce un iteratore

Quindi, tutte le collezioni devono avere un iteratore associato

---

Gerarchia degli iteratori

Esistono due interfacce:



`Iterator`
: un cursore per fare la scansione di un insieme non necessariamente ordinato

`ListIterator`
: un iteratore di una classe che implementa `List`, ossia un insieme ordinato

---

Metodi in più di `ListIterator`

Un oggetto `ListIterator` è un cursore per un oggetto che implementa `List`

Consente di andare avanti e indietro

Consente di inserire un elemento nella posizione corrente

`hasPrevious()`
: ritorna `True` se esiste un elemento precedente nella lista

`Object previous()`
: ritorna l'elemento precedente della lista

`void add(Object o)`
: inserisce un elemento

`void set(Object o)`
: rimpiazza l'elemento con quello passato

Metodi opzionali

Nelle interfacce, viene spesso specificato che un metodo è opzionale

Naturalmente, **deve** essere implementato, altrimenti viene dato errore

linguaggio
: una classe che implementa una interfaccia deve implementare tutti i suoi metodi

contratto

se un metodo è opzionale, la sua implementazione può semplicemente lanciare una eccezione:

```
throw new UnsupportedOperationException("Operation not supported");
```

Esempio

L'interfaccia `Collection` contiene un metodo `add` opzionale

Ogni classe che implementa `Collection` lo deve contenere (altrimenti il compilatore dà errore)

Dal momento che è opzionale, si può anche implementare in questo modo:

```
class Prova implements Collection {
  ...
  public boolean add(Object o) {
    throw new UnsupportedOperationException("Add operation not supported");
  }
}
```