Progettazione del Software

Programmazione in Java (4)

The Collections Framework

Domenico Fabio Savo

Dipartimento di Ingegneria Informatica, Automatica e Gestionale Antonio Ruberti

Sapienza Università di Roma

Le slide di questo corso sono il frutto di una rielaborazione di analogo materiale redatto da Marco Cadoli, Giuseppe De Giacomo, Maurizio Lenzerini e Domenico Lembo

Introduzione al Java Collections Framework

Il **Java Collections Framework** è una libreria formata da un insieme di **interfacce** e di **classi** che le implementano per lavorare con gruppi di oggetti.

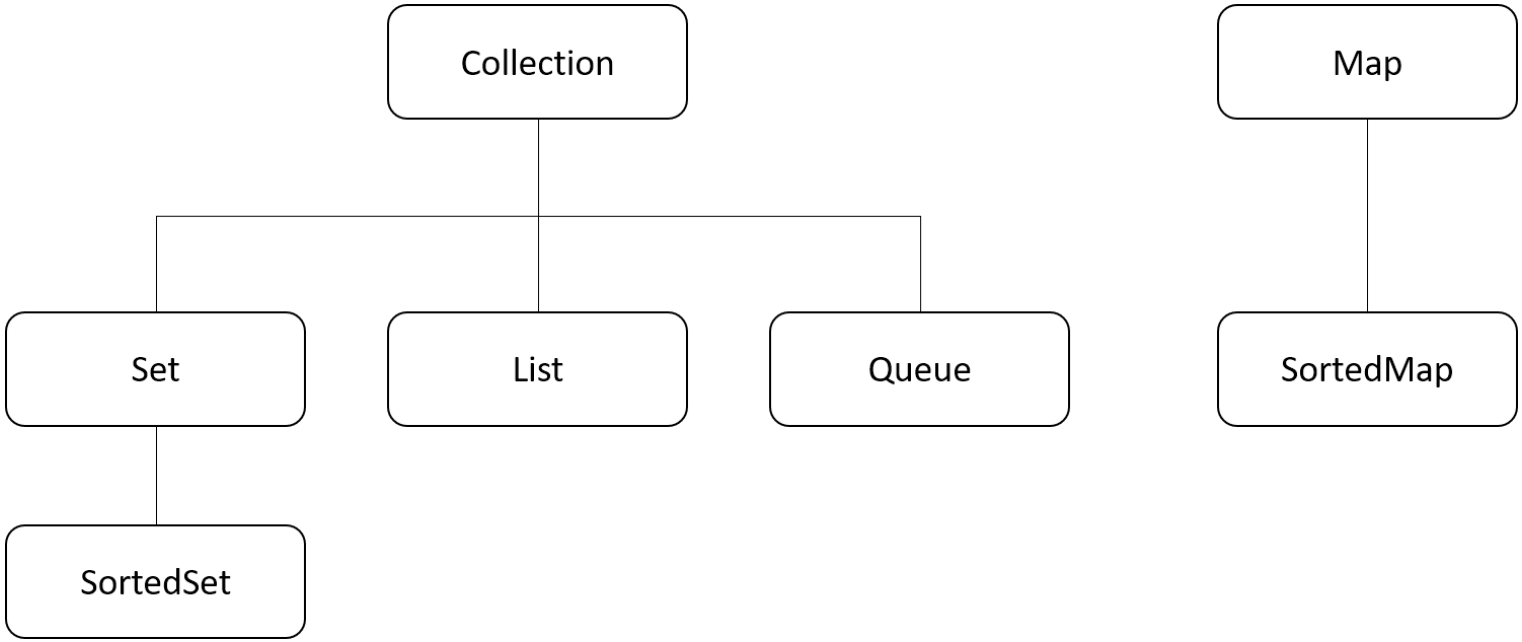
- Le interfacce e le classi del **Collections Framework** si trovano nel package `java.util`
- Il **Collections Framework** comprende:
  - **Interfacce**: rappresentano vari tipi di collezioni di uso comune.
  - **Implementazioni**: sono classi concrete che implementano le interfacce di cui sopra, utilizzando strutture dati efficienti (vedi corso di Algoritmi e Strutture Dati).
  - **Algoritmi**: funzioni che realizzano algoritmi di uso comune, quali algoritmi di ricerca e di ordinamento su oggetti che implementano le interfacce del **Collections Framework**.

Java Collections Framework (cont.)

Perchè usare il **Collections Framework**?

- **Generalità**: permette di modificare l'implementazione di una collezione senza modificare i clienti.
- **Interoperabilità**: permette di utilizzare (e farsi utilizzare da) codice realizzato indipendentemente dal nostro.
- **Efficienza**: le classi che realizzano le collezioni sono ottimizzate per avere prestazioni particolarmente buone (vedi corso di Algoritmi e Strutture Dati).

Interfacce del Collections Framework



In queste slide ci occuperemo solo di `Collection`, `Set`, e `List`.

Collections Framework ed i Generics

A partire dal JSE 1.5, tutte le interfacce del collections framework utilizzano i cosiddetti "generic", risultando in tal modo parametrizzate rispetto alla classe a cui appartengono gli oggetti gestiti dalla collezione.

Per esempio, la dichiarazione dell'interfaccia `Collection` è la seguente

```
public interface Collection<E>...
```

La `<E>` dice che l'interfaccia è generic (ed `E` indica un classe generica, che verrà definita in fase di dichiarazione della collezione). Quando si dichiara una istanza di `Collection` si può (ma è vivamente consigliato farlo) dichiarare il tipo di oggetti contenuti nella collezione (ad es., collezione di stringhe, o di numeri di telefono, ecc....).

```
Collection<String> rubricaTel;
```

Specificare il tipo consente al compilatore di verificare (a tempo di compilazione) che il tipo di oggetto inserito nella collezione sia corretto, riducendo in tal modo errori altrimenti visibili solo a tempo di esecuzione.

Interfaccia Collection

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

  // Bulk Operations
  boolean containsAll(Collection<?> c);
  boolean addAll(Collection<? extends E> c);//Optional
  boolean removeAll(Collection<?> c);       //Optional
  boolean retainAll(Collection<?> c);       //Optional
  void clear();                             //Optional

  // Array Operations
  Object[] toArray();
  <T> T[] toArray(T[] a);
}
```

L'interfaccia specifica

- Operazioni di base quali inserimento, cancellazione, ricerca di un elemento nella collezione
- Operazioni che lavorano su intere collezioni quali l'inserimento, la cancellazione la ricerca di collezioni di elementi
- Operazioni per trasformare il contenuto della collezione in un array.
- Operazioni **"opzionali"** che lanciano `UnsupportedOperationException` se non supportati da una data implementazione dell'interfaccia.

Interfaccia Set

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

  // Bulk Operations
  boolean containsAll(Collection<?> c);
  boolean addAll(Collection<? extends E> c);//Optional
  boolean removeAll(Collection<?> c);       //Optional
  boolean retainAll(Collection<?> c);       //Optional
  void clear();                             //Optional

  // Array Operations
  Object[] toArray();
  <T> T[] toArray(T[] a);
}
```

- Estende `Collection` ed è usato per rappresentare insiemi
- Non contiene altre dichiarazioni di metodi che non siano già presenti in `Collection`
- Non permette di avere elementi duplicati (a differenza di `Collection`)
- `equals` e `hashCode` sono ridefinite in modo che due `Set` sono uguali se contengono gli stessi elementi
- le operazioni "bulk" corrispondono a:
  - `s1.containsAll(s2)` $\Rightarrow$ $S_1 \subseteq S_2$
  - `s1.addAll(s2)` $\Rightarrow$ $S_1 \cup S_2$
  - `s1.removeAll(s2)` $\Rightarrow$ $S_1 \setminus S_2$
  - `s1.retainAll(s2)` $\Rightarrow$ $S_1 \cap S_2$

Iterator

- Un **iteratore** è un oggetto che rappresenta un cursore con il quale scandire una collezione alla quale è associato.
- un iteratore è sempre associato ad un oggetto collezione.
- per funzionare, un oggetto iteratore deve essere a conoscenza degli aspetti più nascosti di una classe, quindi la sua realizzazione dipende interamente dalla classe collezione concreta che implementa la collezione.
- `Iterator<E> iterator()` in `Collection` restituisce un iteratore con il quale scandire la collezione oggetto di invocazione.
- `Iterator` è una interfaccia (non una classe). Questa è sufficiente per utilizzare tutte le funzionalità dell'iteratore senza doverne conoscere alcun dettaglio implementativo.

Iterator (cont.)

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

- Un iteratore ha le seguenti funzionalità:
  - `next()` che restituisce l'elemento corrente della collezione, e contemporaneamente sposta il cursore all'elemento successivo;
  - `hasNext()` che verifica se il cursore ha ancora un successore o se si è raggiunto la fine della collezione;
  - `remove()` che elimina l'elemento restituito dall'ultima invocazione di `next()`;
  - `remove()` è opzionale perchè in certe collezioni non si vuole mettere a disposizione del cliente funzioni che modifichino la collezione durante la scansione dei suoi elementi (come apputo fà `remove()`)

Uso di un Iterator

Un iteratore va usato per scandire la collezione come segue:

```
Collection<E> c = ...      //collezione dove memorizziamo oggetti istanze di String
...
Iterator<E> it = c.iterator();
while (it.hasNext()) {          //finche' il cursore non e' all'ultimo elemento
   E e = it.next();             //  poni l'elemento corrente in e ed avanza
   ...                          //  processa l'elemento corrente (denotato da e)
}
```

Si noti che l'iteratore non ha alcuna funzione che lo "resetti":

- una volta iniziata la scansione, non si può fare tornare indietro l'iteratore;
- una volta finita la scansione, l'iteratore non è più utilizzabile.

Interfaccia List

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    E set(int index, E element);    //optional
    boolean add(E element);         //optional
    void add(int index, E element); //optional
    E remove(int index);            //optional
    boolean addAll(int index,
        Collection<? extends E> c); //optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

- `List` estende `Collection`
- `List` serve a rappresentare il tipo **sequenza** (o lista)
- `List` può permettere di avere elementi duplicati (come `Collection`)
- `List`, oltre alle operazioni ereditate dal `Collection`, include operazioni per:
  - accesso in base alla posizione
  - restituzione della posizione di un oggetto
  - restituzione di sottosequenze
  - scansione bidirezionale della lista (mediante `ListIterator`)

ListIterator

`List` fornisce oltre all'`Iterator` di tutte le `Collection` un iteratore più potente che è in grado di scandire la lista sia in avanti che indietro. Questo iteratore è specificato dall'interfaccia `ListIterator`

```
public interface ListIterator<E> extends Iterator<E> {
//  boolean hasNext();
//  E next();

    boolean hasPrevious();
    E previous();

    int nextIndex();
    int previousIndex();

//  void remove();          // Optional
    void set(E e);      // Optional
    void add(E e);      // Optional
}
```

- Include le funzionalità di Iterator;
- Supporta la scansione inversa della lista (`hasPrevious()` e `previous()` analoghi a `hasNext()` e `next()`);
- Restituisce la posizione dell'iteratore nella lista (`nextIndex()` e `previousIndex()`);
- Permette la sostituzione dell'elemento corrente nella lista (`set()`);
- Permette l'inserimento di un elemento nella lista (`add()`).

ListIterator (cont.)

- `previous()` restituisce l'elemento precedente della lista, e contemporaneamente sposta il cursore all'indietro.

- `hasPrevious()` verifica se il cursore ha ancora un predecessore o si è raggiunto l'inizio della lista.

- `nextIndex()` e `previousIndex()` restituiscono l'indice dell'elemento che sarebbe restituito da `next()` e `previous()` rispettivamente (ma non spostano il cursore).

  All'inizio della lista (quando `hasPrevious()==false`), `previousIndex()` restituisce -1, mente alla fine della lista (quando `hasNext()==false`), `nextIndex()` retituisce `list.size()` (gli elementi sono indicizzati come al solito a partire da 0 fino a `list.size()-1`).

- `set(o)` pone pari ad `o` l'elemento nella posizione corrente (che è la posizione che sarebbe restituita da una chiamata a `next()`).

- `add(o)` aggiunge l'oggetto `o` alla lista nella posizione **precedente** a quella corrente (che è la posizione che sarebbe restituita da una chiamata a `previous()`).

Uso di ListIterator

`ListIterator` può essere usato come un `Iterator` ...

```
List<E> c = ...                    //lista dove memorizziamo oggetti istanze di E
...
ListIterator<E> it = c.listIterator();
while (it.hasNext()) {             //finche' il cursore non e' all'ultimo elemento
   E e = it.next();                //  poni l'elemento corrente in e ed avanza
   ...                             //  processa l'elemento corrente (denotato da e)
}
```

... ma anche per attraversare la lista all'indietro:

```
List<E> c = ...                     //lista dove memorizziamo oggetti istanze di E
...
ListIterator<E> it = c.listIterator(c.size());
while (it.hasPrevious()) {          //finche' il cursore non e' al primo elemento
   E e = it.previous();             //  poni l'elemento precedente in e ed indietreggia
   ...                              //  processa l'elemento denotato da e
}
```

Si noti che `ListIterator listIterator(int i)` in `List` permette di disporre inizialmente il cursore a qualsiasi posizione nella lista.

Implementazioni nel Collections Framework

Ciascuna delle interfacce del Collections Framework è implementata da almeno una classe predefinita che è realizza utilizzando le strutture dati più efficienti per il determinato tipo di collezione *(vedi corso di Algoritmi e Strutture Dati)*.

Queste classi realizzano tutti (e, essenzialmente, soli) i metodi richiesti dall'interfaccia che implementano. Inoltre esse sono dotate di costruttori senza argomenti e ridefiniscono opportunamente `equals()` e `clone()`.

Implementazioni di Collection

- `Collection`: nessuna implementazione specifica.
- `Set`: implementata dalla classe `HashSet` che è basata sull'uso di una **tavola hash**, costo della ricerca, inserimento, e cancellazione pari a $O(1)$ *(vedi corso di Algoritmi e Strutture Dati)*.
- `List`: implementata dalle classi
  - `ArrayList` che è basata su un **array dinamico**, costo della ricerca pari a $O(1)$, inserimento e cancellazione $O(n)$;
  - `LinkedList` che è basata su una **lista doppia** (con riferimento al successore ed al predecessore), costo della ricerca, inserimento e cancellazione $O(n)$, ma inserimento/cancellazione in testa, coda, e durante la scansione dell'iteratore $O(1)$;