SAPIENZA Università di Roma, Facoltà di Ingegneria

Corso di

Progettazione del Software - Esercitazioni

Canale A-L

Anno Accademico 2008-2009

Corso di Laurea in Ingegneria Informatica

Fabio Patrizi

Unità 2

Il 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 (collezioni).

- 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 corsi precedenti).
  - **Algoritmi**: funzioni che realizzano algoritmi di uso comune, quali algoritmi di ricerca e di ordinamento su oggetti che implementano le interfacce del **Collections Framework**.

Il Java Collections Framework (cont.)

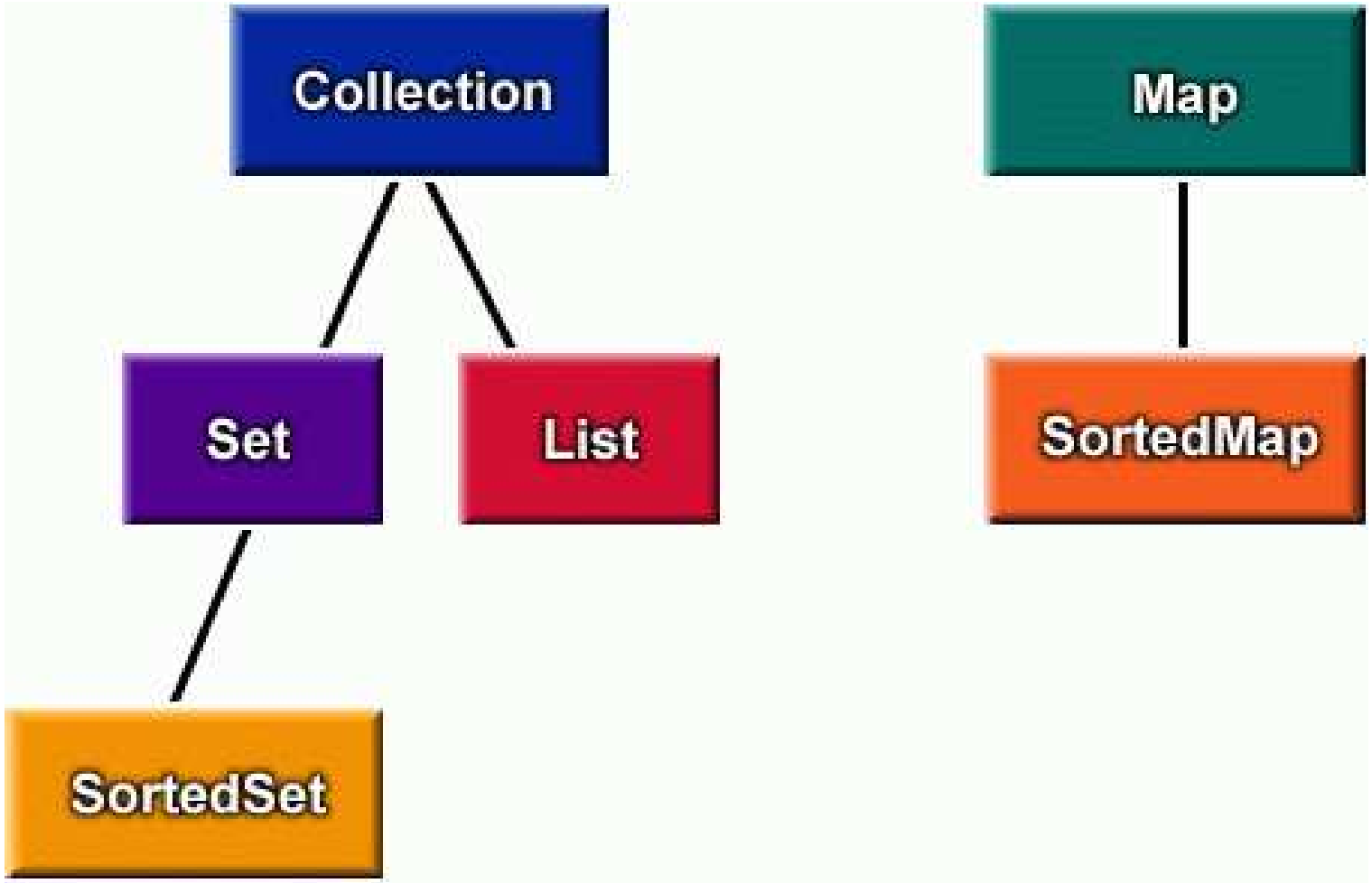
Perchè usare il **Collections Framework**?

- **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 corsi precedenti).

Interfacce del Collections Framework



Interfaccia Collection

```
public interface 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();
    boolean equals(Object o);
    int hashCode();

    // Bulk Operations
    boolean containsAll(Collection <?> c);
    boolean addAll(Collection<? extends E> c); // Opt.
    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

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

Esempio di uso di Collection

- Per ora non possiamo/sappiamo creare un oggetto di classe Collection (è un'interfaccia!)

- ma se qualcuno ce lo fornisse (come oggetto di una classe che implementa Collection), sapremmo usarlo:

```
Collection <String> miaColl = ...
/* NOTA: <String> definisce il tipo degli elementi in miaColl.
miaColl e' una "collezione di String" */

miaColl.add("Ciao"); // Aggiunge un oggetto di tipo String a miaColl
miaColl.clear(); // Svuota miaColl
String[] mioArray; // Crea un riferimento ad array di String;
miaColl.toArray(mioArray); // Popola mioArray con gli elementi di miaColl
...
```

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();
    boolean equals(Object o);
    int hashCode();

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

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

- `Set` estende `Collection`
- `Set` non contiene altre dichiarazioni di metodi che non siano già presenti in `Collection`
- `Set` serve a rappresentare il tipo **insieme**
- `Set` non permette di avere elementi duplicati (a differenza di `Collection`)
- le operazioni "bulk" corrispondono a:
  - `s1.containsAll(s2)` $\Rightarrow$ $S_1 \subseteq S_2$
  - `s1.addAll(s2)` $\Rightarrow$ $S_1 \cup S_2$
  - `s1.retainAll(s2)` $\Rightarrow$ $S_1 \cap S_2$

Iterator

- Un **iteratore** è un oggetto che rappresenta il cursore con cui esplorare sequenzialmente la 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.

- `public 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 perché in certi casi non si vogliono mettere a disposizione del cliente metodi che permettano modifiche arbitrarie alla collezione.

Uso di un Iterator

Un iteratore viene usato per esplorare la collezione come segue:

```
Collection <String> c = ... //collezione in cui sono memorizzati oggetti di classe String
...
Iterator it = c.iterator();  // restituisce l'iteratore associato a c
while (it.hasNext()) {       // finche' il cursore non e' all'ultimo elemento
   String s = it.next();     // poni l'elemento corrente in s ed avanza
   System.out.println(s);    // stampa l'elemento corrente (denotato da s)
   ...
}
```

NOTA: Se al posto di `Collection` comparisse `Set` il programma sarebbe ugualmente corretto poiché `Set` è una classe derivata da `Collection`

Uso di un Iterator (cont.)

Si noti che l'iteratore non ha metodi che lo re-inizializzino:

- una volta iniziata la scansione, non si può fare tornare indietro l'iteratore;
- una volta finita la scansione, l'iteratore non è più utilizzabile (se ne deve ottenere uno nuovo).

Interfaccia List

```
public interface List <E> extends Collection <E>{
    /*...
    Metodi ereditati da Collection
    ...*/

    // Positional Access
    E get(int index);
    E set(int index, E element);         // Optional
    void add(int index, E element);  // Optional
    E remove(int index); // Optional
    boolean addAll(int ind, Collection <? extends E> c);//Opt.

    // 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 sotto-sequenze
  - 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>{
   /* Metodi ereditati da Collection:
   boolean hasNext();
   E next();
   void remove();
   */

    boolean hasPrevious();
    E previous();

    int nextIndex();
    int previousIndex();

    void set(E element);     // Optional
    void add(E element);     // 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`, mentre 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.

- `add(o)` aggiunge l'oggetto `o` alla lista nella posizione **precedente** a quella corrente.

Uso di ListIterator

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

```
List <String> c = ...                    //lista dove memorizziamo oggetti istanze di String
...
ListIterator <String> it = c.listIterator();
while (it.hasNext()) {                  //finche' il cursore non e' all'ultimo elemento
   String 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 <String> c = ...                     //lista dove memorizziamo oggetti istanze di E
...
ListIterator <String> it = c.listIterator(c.size());
while (it.hasPrevious()) {              //finche' il cursore non e' al primo elemento
   String 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 la realizza efficientemente.

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 delle Interfacce `Set` e `List`

- `Collection` non ha nessuna implementazione predefinita *diretta*. Tutte le sue implementazioni realizzano interfacce da essa derivate.

- `Set` è implementata dalla classe `HashSet` (`HashSet <E> implements Set <E>`), basata sull'uso di una **tavola hash**. Costo della ricerca, inserimento, e cancellazione sono pari ad $O(1)$ *(vedi corsi precedenti)*.

- `List` è implementata dalle classi:
  - `ArrayList`(`ArrayList <E> implements List <E>`), basata su un **array dinamico**. Costo della ricerca pari a $O(1)$, inserimento e cancellazione $O(n)$;
  - `LinkedList`(`LinkedList <E> implements List <E>`), basata su una **lista doppia** (con riferimento al successore ed al predecessore). Costo della ricerca, inserimento e cancellazione sono pari ad $O(n)$. Inserimento/cancellazione in testa, coda, e durante la scansione dell'iteratore hanno costo $O(1)$.

Uso delle implementazioni di `Set` e `List`

Le implementazioni di `Set` e `List` ci permettono di creare oggetti di tipo `Collection`, `Set` e `List` (oltre, ovviamente, ad `HashSet`, `ArrayList` e `LinkedList`) e, quindi, usarli come mostrato sopra.

Esempi di uso delle implementazioni di `Set` e `List`

Esempio 1:

```
public class Mattonella{/*...*/}
// Crea un HashSet di Mattonella e vi accede come ad una Collection:
Collection <Mattonella> cMatt = new HashSet <Mattonella>();
```

Esempio 2:

```
import java.util.*;
public static void main(String[] args){
    // Crea una LinkedList di String e vi accede come ad una List:
    List <String> listaParole = new LinkedList <String>();

    listaParole.add("Ciao"); // Appende in coda "Ciao"
    listaParole.add("Mondo"); // Appende in coda "Mondo"

    // Accede alla lista con iteratore semplice:
    // (Potremmo anche usare ListIterator ed avere accesso bidirezionale)
    Iterator<String> listaParoleIt = listaParole.iterator();

    while(listaParoleIt.hasNext()){// Scorre la lista e ne stampa il contenuto
        System.out.println(listaParoleIt.next());
    }
    // stampa: "Ciao Mondo", andando a capo dopo ogni parola
}
```