

Corso di

Progettazione del Software

Anno Accademico 2003-2004

Corso di Laurea in Ingegneria Gestionale

Prof. Giuseppe De Giacomo & Prof. Monica Scannapieco

Java Collections Framework

1

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 (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 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**.

2

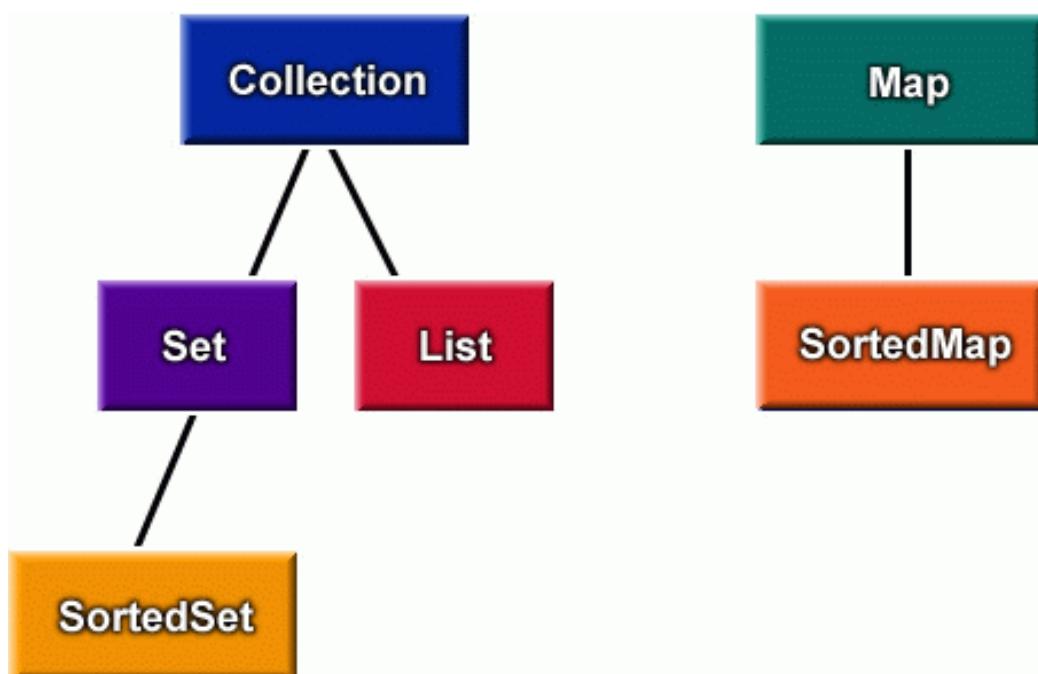
Introduzione al Java Collections Framework

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).

3

Interfacce del Collections Framework



4

Interfaccia Collection

L'interfaccia specifica

```
public interface Collection {  
    // Basic Operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(Object element);    // Optional  
    boolean remove(Object element); // Optional  
    Iterator iterator();  
  
    // Bulk Operations  
    boolean containsAll(Collection c);  
    boolean addAll(Collection c);    // Optional  
    boolean removeAll(Collection c); // Optional  
    boolean retainAll(Collection c); // Optional  
    void clear();                  // Optional  
  
    // Array Operations  
    Object[] toArray();  
    Object[] toArray(Object[] a);  
}
```

- 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.

5

Interfaccia Set

```
public interface Set extends Collection {  
/*  
    // Basic Operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(Object element);    // Optional  
    boolean remove(Object element); // Optional  
    Iterator iterator();  
  
    // Bulk Operations  
    boolean containsAll(Collection c);  
    boolean addAll(Collection c);    // Optional  
    boolean removeAll(Collection c); // Optional  
    boolean retainAll(Collection c); // Optional  
    void clear();                  // Optional  
  
    // Array Operations  
    Object[] toArray();  
    Object[] toArray(Object[] 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:
 - $s1.containsAll(s2) \Rightarrow S_1 \subseteq S_2$
 - $s1.addAll(s2) \Rightarrow S_1 \cup S_2$
 - $s1.retainAll(s2) \Rightarrow S_1 \cap S_2$

6

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.
- `public Iterator 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.

7

Iterator (cont.)

- 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 appunto fa `remove()`)

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

8

Uso di un Iterator

Un iteratore va usato per scandire la collezione come segue:

```
Collection c = ...           //collezione dove memorizziamo oggetti istanze di E
...
Iterator it = c.iterator();
while (it.hasNext()) {        //finche' il cursore non e' all'ultimo elemento
    E 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.

9

Una realizzazione dell'interfaccia Set

Struttura base di una classe

```
public class XXX {
    // campi dati (rappresentazione degli oggetti XXX)
    // costruttori
    // funzioni proprie della classe
    // funzioni speciali ereditate da Object
    // funzioni non pubbliche ausiliarie
}
```

10

Una realizzazione dell'interfaccia Set (cont.)

Implementazione dell'interfaccia Set

```
import java.util.*;  
  
public class InsiemeArray implements Set {  
    // campi dati  
    // costruttori  
  
    // funzioni proprie della classe  
    // (realizzazione delle funzioni di Set)  
  
    // basic operations  
    public int size() { ... }  
    public boolean isEmpty() { ... }  
    public boolean contains(Object e) { ... }  
    public boolean add(Object e) { ... }  
    public boolean remove(Object e) { ... }  
    public Iterator iterator() { ... }  
  
    // bulk Operations  
    public boolean containsAll(Collection c) { ... }  
    public boolean addAll(Collection c){ // opzionale - non supportata  
        throw new UnsupportedOperationException("addAll() non e' supportata");  
    }  
    public boolean removeAll(Collection c) { // opzionale - non supportata  
        throw new UnsupportedOperationException("removeAll() non e' supportata");  
    }  
    public boolean retainAll(Collection c) { // opzionale - non supportata  
        throw new UnsupportedOperationException("retainAll() non e' supportata");  
    }  
    public void clear() { // opzionale - non supportata  
        throw new UnsupportedOperationException("clear() non e' supportata");  
    }  
    // array operations  
    public Object[] toArray() { ... }  
    public Object[] toArray(Object[] a) { ... }  
  
    // funzioni speciali ereditate da Object  
    // funzioni ausiliarie  
}
```

Una realizzazione dell'interfaccia Set

Rappresentazione degli oggetti InsiemeArray basata su array “dinamici”.

```
public class InsiemeArray implements Set {  
    // campi dati  
    protected Object[] array;  
    protected static final int dimInit = 10; //dim. iniz. array  
  
    protected int cardinalita;  
  
    // costruttori  
  
    // funzioni proprie della classe  
    // (realizzazione delle funzioni di Set)  
    ...  
}
```

12

Una realizzazione dell'interfaccia Set

Costruttori per InsiemeArray

```
public class InsiemeArray implements Set {  
    // campi dati  
    protected Object[] array;  
    protected static final int dimInit = 10; //dim. iniz. array  
  
    protected int cardinalita;  
  
    // costruttori  
    public InsiemeArray() {  
        array = new Object[dimInit];  
        cardinalita = 0;  
    }  
  
    ...  
}
```

13

Una realizzazione dell'interfaccia Set

Funzioni speciali ereditate da Object

```
public class InsiemeArray implements Set, Cloneable {  
    // campi dati  
    ...  
  
    // costruttori  
    ...  
  
    // funzioni proprie della classe  
    // (realizzazione delle funzioni di Set)  
    ...  
  
    // funzioni speciali ereditate da Object  
    public boolean equals(Object o) {  
        //verifica uguaglianza profonda  
    }  
  
    public Object clone() {  
        //verifica copia profonda  
    }  
  
    public String toString() { ... }  
}
```

14

Una realizzazione dell'interfaccia Set

Realizzazione dei singoli metodi (vedi codice allegato):

- equals(), clone(), toString()
- size(), isEmpty(), contains()
- add(), remove()

15

Una realizzazione dell'interfaccia Set

Realizzazione della funzione: iterator()

```
public class InsiemeArray implements Set, Cloneable {  
    // campi dati  
    . . .  
    // funzioni proprie della classe  
    // (realizzazione delle funzioni di Set)  
    // basic operations  
    . . .  
    public Iterator iterator() {  
        return new IteratorInsiemeArray(this); //nota!  
    }  
    . . .  
}
```

16

Interfaccia Iterator

L'interfaccia Iterator

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove(); // Optional <- noi non lo realizzeremo mai  
}
```

Esempio di uso dell'interfaccia Iterator

```
public class ClienteDiSet {  
    . . .  
    public static void StampaElementi(Set s) {  
        Iterator it = s.iterator();  
        while(it.hasNext()) {  
            Object o = it.next();  
            System.out.println(o);  
        }  
    }  
    . . .  
}
```

17

Realizzazione dell'interfaccia Iterator

Realizziamo iteratore per InsiemeArray

```
// Quanto segue deve stare nello stesso package di InsiemeArray

package insiemearray;

import java.util.*;

class IteratorInsiemeArray implements Iterator {
// nota non e' pubblica, cioe' ne impediamo l'uso diretto da parte
// dei clienti (che quindi possono usare solo l'interfaccia Iterator)

    private InsiemeArray insiemeArray;
    private int indice;

    public IteratorInsiemeArray(InsiemeArray ia) {
        insiemeArray = ia;
        indice = 0;
    }

    // Realizzazione funzioni di Itarator
    public boolean hasNext() {
        return indice < insiemeArray.cardinalita;
    }

    //nota accessibile perche' nello stesso package!!!
}

public Object next() {
    Object e = insiemeArray.array[indice];
    //nota accessibile perche' nello stesso package!!!
    indice++;
    return e;
}

public void remove() { // non non la supportiamo
    throw new UnsupportedOperationException("remove() non e' supportata");
}
}
```

Collezioni omogenee

L'implementazione InsiemeArray di Set, cosi' come tutte le implementazioni predefinite del Collections Framework, realizzano collezioni **disomogenee**, cioe' i cui elementi possono essere di tipi diversi (es. In prima posizione possiamo avere una String, in seconda un Integer, ecc).

Come possiamo realizzare collezioni **omogenee**, cioe' i cui elementi siano tutti dello stesso tipo?

Imporre che una collezione sia omogenea in modo statico (cioe' a tempo di compilazione) non si puo' in Java!

Ma si puo' imporlo in modo dinamico (cioe' a tempo di esecuzione):

- dobbiamo memorizzare nella classe il tipo degli elementi,
- dobbiamo verificare durante l'inserimento che gli oggetti che inseriamo siano del tipo giusto.

19

Insieme di elementi omogenei

Costruiamo una classe InsiemeArrayOmogeneo che implementa Set, ma realizza insiemi omogenei.

Riusiamo l'implementazione InsiemeArray, realizzando InsiemeArrayOmogeneo come una sua classe derivata, dove faremo overriding dei metodi che vanno modificati

```
public class InsiemeArrayOmogeneo extends InsiemeArray {  
    // campi dati  
    protected Class elemClass;  
  
    // costruttori  
    public InsiemeArrayOmogeneo(Class cl) {  
        super();  
        elemClass = cl;  
    }  
    public InsiemeArrayOmogeneo() {  
        this(Object.class);  
    }  
  
    // overriding delle basic operations
```

20

```
public boolean contains(Object e) {
    if (!elemClass.isInstance(e)) return false;
    else return super.contains(e);
}

public boolean add(Object e) {
    if (!elemClass.isInstance(e)) return false;
    else return super.add(e);
}

public boolean remove(Object e) {
    if (!elemClass.isInstance(e)) return false;
    else return super.remove(e);
}

// overriding delle bulk operations: non serve
// overriding delle array operations: non serve
...

// overriding delle funzioni speciali ereditate da Object
public boolean equals(Object o) {
    if (super.equals(o)) {
        InsiemeArrayOmogeneo ins = (InsiemeArrayOmogeneo)o;
        if (elemClass.equals(ins.elemClass)) return true;
        else return false;
    }
    else return false;
}

public Object clone() {
    InsiemeArrayOmogeneo ins = (InsiemeArrayOmogeneo)super.clone();
    return ins; // Nota: clone() di Object si occupa già della copia
               // (superficiale) del campo elemClass.
    // oppure semplicemente: return super.clone();
}
// overriding di toString() non serve
...
```