Università di Roma "La Sapienza", Facoltà di Ingegneria

Corso di

Laboratorio di Programmazione

Anno Accademico 2004-2005

Corso di Laurea in Ingegneria Informatica

Prof. Giuseppe De Giacomo (A–L) & Prof. Paolo Liberatore (M–Z)

Java Collections Framework

Unità 2

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).

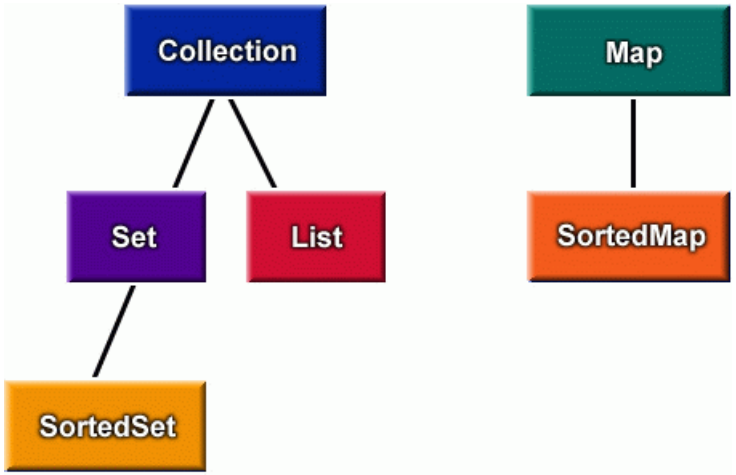
- Le interfacce e le classi del **Collections Framework** si trovano nel package `java.util`
- Il **Collections Framework** comprende:
  - **Interfacce**: rappresentano vari tipi di collezioni di uso comune.
  - **Implementazioni**: sono classi concrete che implementano le interfacce di cui sopra, utilizzando 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**.

Introduzione al 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



Interfaccia Collection

```
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);
}
```

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 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)` $\Rightarrow$ $S_1 \cap S_2$

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'interatore senza doverne conoscere alcun dettaglio implementativo.

Iterator (cont.)

```
public interface Iterator {
    boolean hasNext();
    Object 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 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.

Interfaccia List

```
public interface List extends Collection {
    // Positional Access
    Object get(int index);
    Object set(int index, Object element);      // Optional
    void add(int index, Object element);        // Optional
    Object remove(int index);                   // Optional
    boolean addAll(int index, Collection c);    // Optional

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

    // Iteration
    ListIterator listIterator();
    ListIterator listIterator(int index);

    // Range-view
    List 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 extends Iterator {
//  boolean hasNext();
//  Object next();

    boolean hasPrevious();
    Object previous();

    int nextIndex();
    int previousIndex();

//  void remove();          // Optional
    void set(Object o);     // Optional
    void add(Object o);     // 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.

- `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 c = ...                  //lista dove memorizziamo oggetti istanze di E
...
ListIterator it = c.listIterator();
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)
}
```

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

```
List c = ...                  //lista dove memorizziamo oggetti istanze di E
...
ListIterator it = c.listIterator(c.size());
while (it.hasPrevious()) {    //finche' il cursore non e' al primo elemento
   E 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.

Mapping

Un mapping è una struttura dati che serve a memorizzare coppie **(chiave,informazione)**

- La **chiave** serve ad accedere alla coppia memorizzata nel mapping

  Nella collezione non possono esserci due coppie con la stessa chiave.

- Il campo **informazione** serve a memorizzare informazioni di interesse da associare alla chiave.

Nota: nel Collections Framework sia la chiave che l'informazione (chiamata "valore") sono istanze qualsiasi di `Object`.

Mapping (cont.)

Esempi di mapping:

- dizionario - coppie (parola, significato)
- elenco del telefono - coppie (nominativo, indirizzo e telefono)
- funzione discreta - coppie (x,y)
- ...

Interfaccia Map

```
public interface Map {
    // Basic Operations
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk Operations
    void putAll(Map t);
    void clear();

    // Collection Views
    public Set keySet();
    public Collection values();
    public Set entrySet();

    // Interface for entrySet elements
    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```

- `Map` **non** estende `Collection`
- `Map` fornisce funzioni necessarie per accedere ai suoi elementi attraverso la chiave (`put()` –che si comporta come `set()` se la chiave è già presente– `get()`, `remove()`, `containsKey()`).
- `Map` permette di restituire
  - l'insieme delle sue chiavi
  - la collezione dei suoi valori (alcuni valori possono essere ripetuti)
  - l'insieme delle coppie *(chiave,valore)*
  - per accedere a queste coppie si fa uso dell'interfaccia `Map.Entry`
- `Map` ha altre funzioni di uso comune (`containsValue()`, tipicamente molto meno efficiente dell'accesso per chiave, `size()`, `isEmpty()`, `putAll()`, `clear()`).

Collezioni ordinate

Il Java Collections Framework prevede anche la possibilità (molto comune nella pratica, di definire collezioni ordinate.

In particolare esso prevede due interfacce per collezioni ordinate:

- `SortedSet`: per rappresentare insiemi di oggetti ordinati (non sono ammessi oggetti ripetuti).
- `SortedMap`: per rappresentare mapping ordinati per chiave.

Interfaccia SortedSet

```
public interface SortedSet extends Set {
    // Range-view
    SortedSet subSet(Object from, Object to);
    SortedSet headSet(Object to);
    SortedSet tailSet(Object from);

    // Endpoints
    Object first();
    Object last();

    // Comparator access
    Comparator comparator();
}
```

- `SortedSet` rappresenta una collezione di oggetti **senza ripetizione**, come `Set`, ma **ordinati**.
- `SortedSet` estende `Set`, e tutte le operazioni ereditate da `Set` si comportano in `SortedSet` come in `Set`.
- L'iteratore restituito da `iterator()` scandisce l'insieme rispettando l'**ordinamento**, e `toArray()` restituisce un array con gli elementi dell'insieme **ordinati**.
- `subSet()` restituisce l'insieme ordinato formato dagli elementi dell'insieme oggetto di invocazione che sono **più grandi o uguali** a `from` e **strettamente più piccoli** di `to`.
- `headSet()` restituisce l'insieme ordinato formato dagli elementi dell'insieme oggetto di invocazione che sono **strettamente più piccoli** di `to`.
- `subset()` restituisce l'insieme ordinato formato dagli elementi dell'insieme oggetto di invocazione che sono **più grandi o uguali** a `from`.
- `first()` e `last()` restituiscono rispettivamente l'oggetto piú piccolo e l'oggetto più grande dell'insieme.
- `comparator()` restituisce l'oggetto **comparatore** usato oppure `null` se non l'insieme non usa un comparatore specifico (vedi dopo).

Interfaccia SortedMap

```
public interface SortedMap extends Map {

  SortedMap subMap(Object fromKey, Object toKey);
  SortedMap headMap(Object toKey);
  SortedMap tailMap(Object fromKey);

  Object first();
  Object last();

  Comparator comparator();
}
```

- `SortedSet` rappresenta un mapping di oggetti come `Map`, ma con le **chiavi ordinate**.
- `SortedMap` estende `Map`, e tutte le operazioni ereditate da `Map` si comportano in `SortedMap` come in `Map`.
- L'iteratore restituito da `iterator()` dell'insieme restituito da `keySet()`, `entrySet()`, scandisce l'insieme rispettando l'**ordinamento selle chiavi**, e `toArray()` restituisce un array con gli elementi dell'insieme **ordinati**.
- `subMap()` restituisce un mapping ordinato formato dalle "entry" del mapping oggetto di invocazione le cui chiavi sono **più grandi o uguali** a `fromKey` e **strettamente più piccole** di `toKey`.
- `headMap()` restituisce un mapping ordinato formato dalle "entry" del mapping oggetto di invocazione le cui chiavi **strettamente più piccole** di `toKey`.
- `subMap()` restituisce un mapping ordinato formato dalle "entry" del mapping oggetto di invocazione le cui chiavi sono **più grandi o uguali** a `fromKey`.
- `first()` e `last()` restituiscono rispettivamente la chiave piú piccola e la chiave più grande del mapping.
- `comparator()` restituisce l'oggetto **comparatore** usato oppure `null` se il mapping non usa un comparatore specifico (vedi dopo).

Ordinamento usato da SortedSet e SortedMap

Per mantenere ordinata la collezione `SortedSet` e `SortedMap` possono fare uso:

- **ordinamento definito dall'interfaccia** `Comparable` (detto, "ordinamento naturale" nella documentazione Java) per gli oggetti che implementano tale interfaccia;
- **ordinamento definito da un oggetto comparatore** per tutti gli oggetti (indipendentemente dal fatto che implementino o meno `Comparable`.

Interfaccia Comparable

L'interfacia `Comparable` è definita come segue:

```
public interface Comparable {
    public int compareTo(Object o);
}
```

Il metodo `compareTo()` confronta l'oggetto di invocazione con l'oggetto passato come parametro, restituendo un intero negativo (tipicamente -1), 0, o un intero positivo (tipicamente 1) rispettivamente se l'oggetto di invocazione è più piccolo, uguale, o più grande dell'oggetto passato come parametro. Se l'oggetto passato come parametro non è comparabile con l'oggetto di invocazione, il metodo lancia una `ClassCastException`.

L'interfaccia `Comparable` è parte del package `java.util`.

Molte classi predefinite implementano l'interfaccia `Comparable`, in particolare la classe `String`, le classi wrapper dei tipi base (`Integer`, `Double`, ecc.), la classe `Date`, ecc.

Interfaccia Comparator

L'interfacia `Comparator` serve a definire oggetti comparatori per confrontare oggetti di classi che non implementano `Comparable`, oppure per utilizzare un ordinamento diverso da quello definito da `compareTo()` di `Comparable`.

L'interfaccia `Comparator` è definita come segue:

```
public interface Comparator {
    int compare(Object o1, Object o2);
}
```

Il metodo `compare()` confronta l'oggetto denotato da `o1` con l'oggetto denotato da `o2`, restituendo un intero negativo (tipicamente -1), 0, o un intero positivo (tipicamente 1) rispettivamente se `o1` è più piccolo, uguale, o più grande di `o2`. Se uno dei due oggetti denotati da `o1` e `o2` è inappropriato rispetto al comparatore (e.g., non è del tipo giusto), il metodo lancia una `ClassCastException`.

L'interfaccia `Comparator` è parte del package `java.util`.

Tipo di ordinamento di una collezione ordinata

Le classi che implementano `SortedSet` o `SortedMap` sono tipicamente dotate di due costruttori:

- **Costruttore senza argomenti**: che basa l'ordinamento degli oggetti nella collezione sul fatto che questi implementino l'interfaccia `comparable` e quindi siano dotati della funzione `compareTo()`; se la collezione è creata con questo costruttore, la funzione `comparator()` restituisce `null`.
- **Costruttore con un argomento di tipo** `Comparator`: che basa l'ordinamento degli oggetti nella collezione sull'ordinamento indotto dalla funzione `compare` del comparatore passato al costruttore; se la collezione è creata con questo costruttore, la funzione `comparator()` restituisce l'oggetto comparatore utilizzato dalla collezione.

Scegliendo il tipo di costruttore da utilizzare per creare la classe collezione definiamo che tipo ordinamento consideriamo sugli oggetti della collezione.

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)*.
- `SortedSet`: implementata dalla classe `TreeSet` che è basata sull'uso di un **albero di ricerca bilanciato**, costo della ricerca, inserimento e cancellazione pari a $O(log(n))$ *(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 $O(n)$ *(vedi dopo)*;
  - `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 $O(1)$ *(vedi dopo)*.

Implementazioni di Map

- `Map`: implementata dalla classe `HashMap` che è basata sull'uso di una **tavola hash**, costo della ricerca, inserimento e cancellazione per chiave pari a $O(1)$ *(vedi corso di Algoritmi e Strutture Dati)*.
- `SortedMap`: implementata dalla classe `TreeMap` che è basata sull'uso di un **albero di ricerca bilanciato**, costo della ricerca, inserimento e cancellazione per chiave pari a $O(log(n))$ *(vedi corso di Algoritmi e Strutture Dati)*.