

Strutture dati nella piattaforma Java: Java Collection Framework

Leggere cap. 15 di Programmazione di base e avanzata con Java

Sorgente:

Prof. Enrico Denti

Fondamenti di Informatica T-2

Corso di Laurea in Ingegneria Informatica

Universita' di Bologna

STRUTTURE DATI IN JAVA

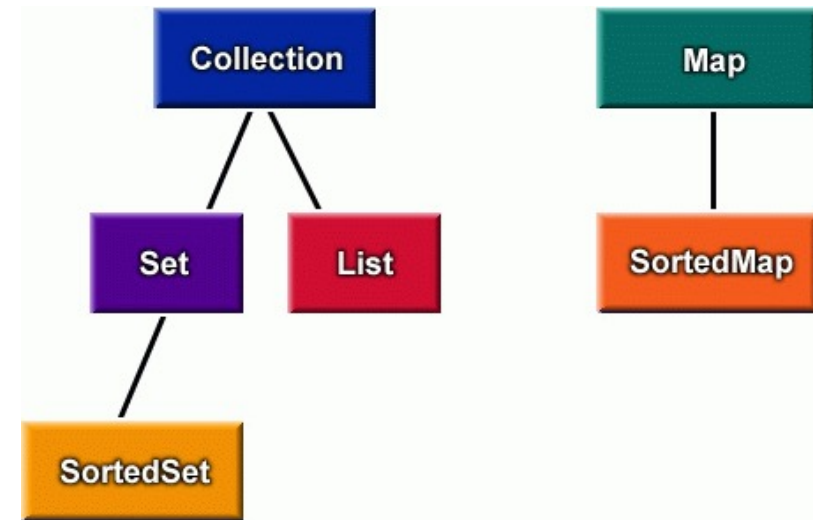
- **Java Collection Framework (JCF)** fornisce il supporto a molte **strutture dati (collezioni di oggetti: liste, insiemi, ...)**, nel quadro di **un'architettura logica globale e uniforme**
 - **interfacce** che definiscono **TIPI DI STRUTTURE DATI** e i necessari **concetti di supporto** (es.: *iteratori*);
 - **una classe Collections** che definisce **algoritmi polimorfi** sotto forma di funzioni statiche, nonché servizi e costanti di uso generale;
 - **classi** che forniscono **implementazioni** dei vari tipi di strutture dati specificati dalle interfacce.
- Obiettivo: **strutture dati per "elementi generici"**

JAVA COLLECTION FRAMEWORK

(package `java.util`)

Interfacce fondamentali

- **Collection**: nessuna ipotesi sul tipo di collezione
- **Set**: introduce l'idea di *insieme* di elementi (quindi, senza duplicati)
- **List**: introduce l'idea di *sequenza*
- **SortedSet**: l'insieme *ordinato*
- **Map**: introduce l'idea di *mappa*, ossia tabella che associa chiavi a valori
- **SortedMap**: una mappa (tabella) *ordinata*



Criteri-guida per la definizione delle interfacce:

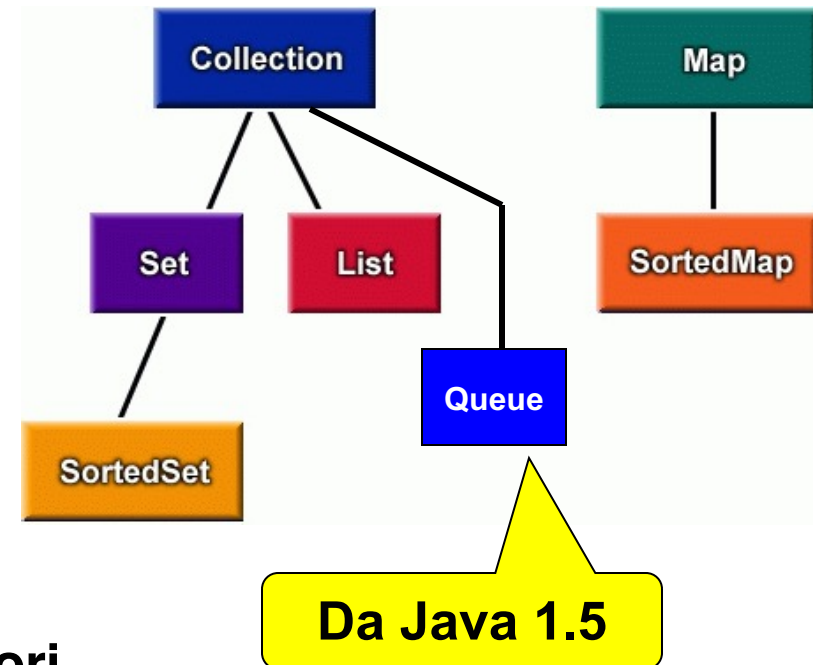
- **Minimalità** – prevedere solo metodi *davvero basilari*...
- **Efficienza** – ...o che *migliorino nettamente le prestazioni* 3

JAVA COLLECTION FRAMEWORK (package `java.util`)

Interfacce fondamentali

- **Collection**: nessuna ipotesi sul tipo di collezione
- **Set**: introduce l'idea di *insieme* di elementi (quindi, senza duplicati)
- **List**: introduce l'idea di *sequenza*
- **SortedSet**: l'insieme *ordinato*
- **Map**: introduce l'idea di *mappa*, ossia tabella che associa chiavi a valori
- **SortedMap**: una mappa (tabella) *ordinata*

- **Queue**: introduce l'idea di *coda di elementi* (non necessariamente operante in modo FIFO: sono "code" anche gli stack.. che operano LIFO!)



L'INTERFACCIA Collection

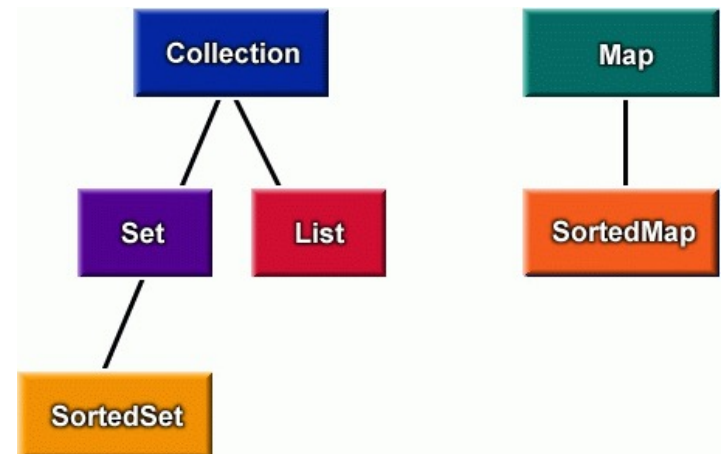
Collection introduce l'idea di **collezione di elementi**

- non si fanno ipotesi sulla natura di tale collezione
 - in particolare, non si dice che sia un insieme o una sequenza, né che ci sia o meno un ordinamento,.. etc
- perciò, ***l'interfaccia di accesso è volutamente generale*** e prevede metodi per :
 - assicurarsi che un elemento sia nella collezione `add(Object o)`
 - rimuovere un elemento dalla collezione. `remove(Object o)`
 - verificare se un elemento è nella collezione. `contains(Object o)`
 - verificare se la collezione è vuota `isEmpty()`
 - sapere la cardinalità della collezione `size()`
 - ottenere un array con gli stessi elementi `toArray()`
 - verificare se due collezioni sono "uguali» `equals(Collection c)`
 - ... e altri ...

L'INTERFACCIA Set

Set estende e specializza Collection introducendo l'idea di **insieme di elementi**

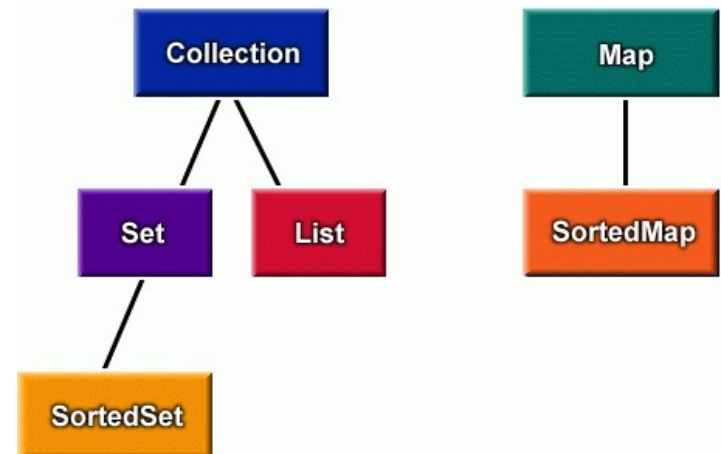
- in quanto insieme, *non ammette elementi duplicati e non ha una nozione di sequenza o di posizione*
- ***l'interfaccia di accesso*** non cambia sintatticamente, ***ma prevede nuovi vincoli al contratto d'uso:***
 - **add** aggiunge un elemento solo se esso non è già presente
 - **equals** assicura che due set siano identici nel senso che $\forall x \in S1, x \in S2$ e viceversa
 - **tutti i costruttori** si impegnano a creare insiemi privi di duplicati



L'INTERFACCIA `List`

`List` estende e specializza `Collection` introducendo l'idea di **sequenza di elementi**

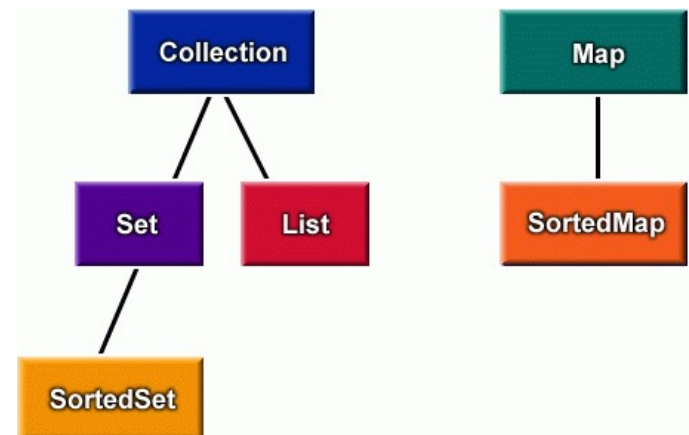
- *tipicamente ammette duplicati*
- in quanto sequenza, ha una nozione di *posizione*
- *l'interfaccia di accesso* aggiunge sia **nuovi vincoli al contratto d'uso**, sia **nuovi metodi per l'accesso posizionale**
 - `add` aggiunge un elemento in fondo alla lista (append)
 - `equals` è vero se gli elementi corrispondenti sono tutti uguali due a due (o sono entrambi null)
 - nuovi metodi `set`, `remove`, `get` accedono alla lista **per posizione**



L'INTERFACCIA SortedSet

SortedSet estende e specializza Set introducendo l'idea di **ordinamento totale** fra gli elementi

- l'ordinamento è quello naturale degli elementi (espresso dalla loro `compareTo`) o quello incapsulato da un `Comparator` fornito all'atto della creazione del `SortedSet`
- ***l'interfaccia di accesso aggiunge metodi*** che sfruttano l'esistenza di un ordinamento totale fra gli elementi:
 - **`first`** e **`last`** restituiscono il primo e l'ultimo elemento nell'ordine
 - **`headSet`**, **`subSet`** e **`tailSet`** restituiscono i *sottoinsiemi ordinati* contenenti rispettivamente i soli elementi minori di quello dato, compresi fra i due dati, e maggiori di quello dato.

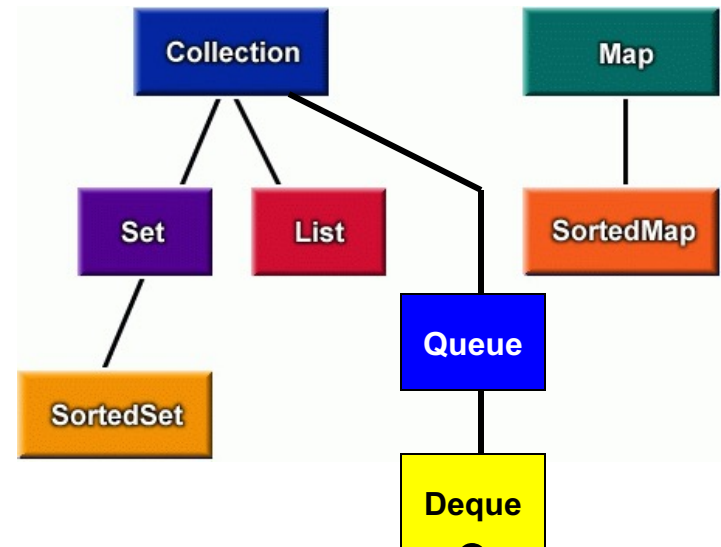


LE INTERFACCE Queue E Deque

Queue (\geq JDK 1.5) **specializza Collection** introducendo l'idea di **coda** di elementi da sottoporre a elaborazione

- *ha una nozione di posizione (testa della coda)*
- *l'interfaccia di accesso si specializza:*
 - **remove** estrae l'elemento "in testa" alla coda, rimuovendolo
 - **element** lo estrae senza rimuoverlo
 - esistono analoghi metodi che, anziché lanciare eccezione in caso di problemi, restituiscono un'indicazione di fallimento

Deque (\geq JDK 1.6) **specializza Queue** con l'idea di **doppia coda** (una coda in cui si possono inserire/togliere elementi da entrambe le estremità)



L'INTERFACCIA Map

Map introduce l'idea di *tabella di elementi, ognuno associato univocamente a una chiave identificativa*.

- in pratica, è una *tabella a due colonne (chiavi, elementi)* in cui i dati della prima colonna (*chiavi*) identificano univocamente la riga

chiave	valore
key1	oggetto1
key2	oggetto2
...	...

Obiettivo: accedere velocemente agli elementi in base alla chiave

- **IDEALMENTE, IN UN TEMPO COSTANTE**: ciò è possibile se si dispone di una *opportuna funzione matematica* che metta in corrispondenza chiavi e valori (*funzione hash*): *data la chiave*, tale funzione *restituisce la posizione in tabella* dell'elemento
- **in alternativa**, si possono predisporre opportuni alberi per guidare il reperimento dell'elemento a partire dalla chiave.

L'INTERFACCIA Map

L'interfaccia di accesso prevede metodi per :

- inserire in tabella una coppia (*chiave*, *elemento*) put
- accedere a un elemento in tabella, data la chiave get
- verificare se una *chiave* è presente in tabella containsKey
- verificare se un *elemento* è presente in tabella containsValue

<i>chiave</i>	<i>valore</i>
key1	oggetto1
key2	oggetto2
...	...

ogg = get(key2)
restituisce *oggetto2*

put(key3, oggetto3)

- Non importa dove viene fisicamente messa la nuova riga
- l'accesso avviene comunque per chiave, in modo efficiente (tempo costante, se possibile)

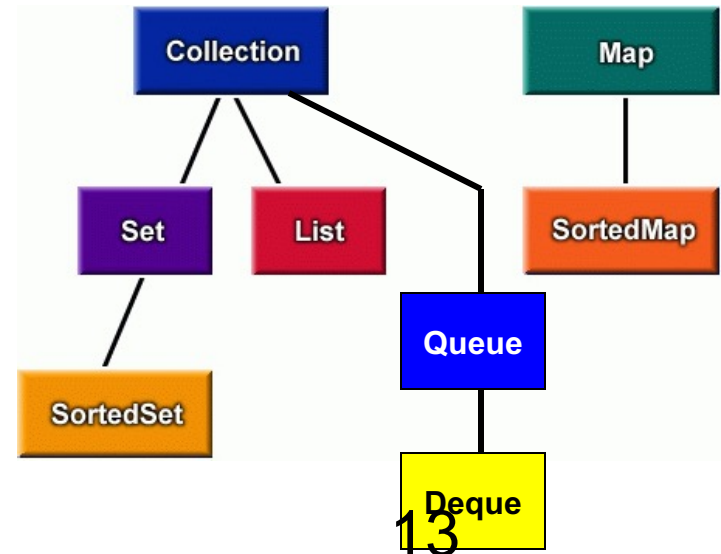
L'INTERFACCIA Map (continua)

- **L'interfaccia di accesso supporta inoltre le cosiddette "Collection views", che estraggono dalla mappa:**
 - TUTTA LA COLONNA CHIAVI: **keySet**
 - TUTTA LA COLONNA VALORI **values**
 - TUTTE LE RIGHE
ovvero tutte le coppie (*chiave, elemento*) **entrySet**
- **Tali metodi restituiscono di fatto *altre collections*, ovvero rispettivamente:**
 - un set (perché le chiavi non ammettono duplicati)
 - una collection (perché sui valori non ci sono ipotesi)
 - un set di Entry (ognuna rappresenta una riga)

L'INTERFACCIA SortedMap

SortedMap estende e specializza Map analogamente a quanto SortedSet fa con Set

- **l'ordinamento è quello naturale delle chiavi (espresso dalla loro compareTo) o quello fornito da un apposito Comparator all'atto della creazione del SortedSet**
- ***l'interfaccia di accesso aggiunge metodi*** che sfruttano l'esistenza di un ordinamento totale fra gli elementi:
 - **firstKey** e **lastKey** restituiscono la prima/ultima chiave nell'ordine
 - **headMap**, **subMap** e **tailMap** restituiscono le sottotabelle con le sole entry le cui chiavi sono minori/comprese/maggiori di quella data.



LA CLASSE Collections

- A completamento dell'architettura logica di JCF, alle interfacce si accompagna la **classe Collections**
- Essa contiene **metodi statici** per collezioni:
 - alcuni incapsulano **algoritmi polimorfi** che operano su qualunque tipo di collezione
 - *ordinamento, ricerca binaria, riempimento, ricerca del minimo e del massimo, sostituzioni, reverse,...*
 - **altri sono "wrapper"** che incapsulano una collezione di un tipo in un'istanza di un altro tipo
- Fornisce inoltre alcune **costanti** :
 - la lista vuota (**EMPTY LIST**)
 - l'insieme vuoto (**EMPTY SET**)
 - la mappa vuota (**EMPTY MAP**)

LA CLASSE Collections

Alcuni algoritmi rilevanti per collezioni qualsiasi:

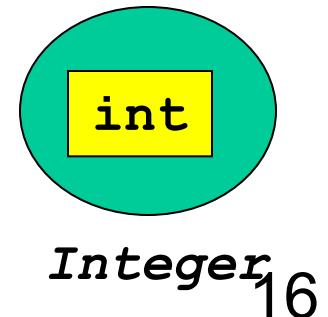
- **sort(List)**: ordina una lista con una versione migliorata di *merge sort* che garantisce tempi dell'ordine di $n \cdot \log(n)$
 - NB: l'implementazione copia la lista in un array e ordina quello, poi lo ricopia nella lista: così facendo, evita il calo di prestazioni a $n^2 \cdot \log(n)$ che si avrebbe tentando di ordinare la lista sul posto.
- **reverse(List)**: inverte l'ordine degli elementi della lista
- **copy(List dest, List src)**: copia una lista nell'altra
- **binarySearch(List, Object)**: cerca l'elemento nella lista ordinata fornita, tramite ricerca binaria.
 - le prestazioni sono ottimali – $\log(n)$ – se la lista permette l'accesso casuale, ossia fornisce un modo per accedere ai vari elementi in tempo circa costante (interfaccia RandomAccess).

TRATTAMENTO DEI TIPI PRIMITIVI

- **PROBLEMA:** *i tipi primitivi* sono i "mattoni elementari" del linguaggio, *ma non sono classi*
 - non derivano da `Object` → *non usabili nella JCF classica*
 - i valori primitivi non sono uniformi agli oggetti !
- **LA CURA:** *incapsularli* in opportuni oggetti
 - l'incapsulamento di un valore primitivo in un opportuno oggetto si chiama **BOXING**
 - l'operazione duale si chiama **UNBOXING**

Il linguaggio offre già le necessarie *classi wrapper*

<code>boolean</code>	<code>Boolean</code>	<code>char</code>	<code>Character</code>
<code>byte</code>	<code>Byte</code>	<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>	<code>long</code>	<code>Long</code>
<code>double</code>	<code>Double</code>	<code>float</code>	<code>Float</code>



JAVA 1.5: BOXING AUTOMATICO

- Da Java 1.5, come già in C#, **boxing e unboxing** sono diventati **automatici**.
- È quindi possibile ***inserire direttamente valori primitivi in strutture dati***, come pure ***effettuare operazioni aritmetiche*** su oggetti incapsulati.

```
List list = new ArrayList();  
list.add(21); // OK da Java 1.5 in poi  
int i = (Integer) list.get(0);
```

```
Integer x = new Integer(23);  
Integer y = new Integer(4);  
Integer z = x + y; // OK da Java 1.5
```

ITERATORI

JCF introduce il concetto di *iteratore* come *mezzo per iterare su una collezione di elementi*

- l'iteratore svolge per la collezione un ruolo analogo a quello di una variabile di ciclo in un array: *garantisce che ogni elemento venga considerato una e una sola volta, indipendentemente dal tipo di collezione e da come essa sia realizzata*
- l'iteratore costituisce dunque un mezzo per "ciclare" in una collezione con una semantica chiara e ben definita, anche se la collezione venisse modificata
- è l'iteratore che rende possibile il nuovo costrutto *for (foreach in C#)*, poiché, mascherando i dettagli, uniforma l'accesso agli elementi di una collezione

ITERATORI

Di fatto, **ogni iteratore** offre:

- un metodo **next** che restituisce *"il prossimo" elemento della collezione*
 - esso garantisce che tutti gli elementi siano prima o poi considerati, senza duplicazioni né esclusioni
- un metodo **hasNext** per sapere se ci sono altri elementi

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

- **Per ottenere un iteratore per una data collezione, basta chiamare su di essa il metodo **iterator****

ITERATORI e NUOVO COSTRUTTO `for`

L'idea di *iteratore* è alla base del **nuovo costrutto `for`** (foreach in C#), in quanto la scrittura:

```
for (type x : coll) { /* operazioni su x */ }
```

equivale a:

```
for (Iterator i = coll.iterator() ; i.hasNext() ; )  
    { /* operazioni su x = i.next() */ }
```

Il nuovo `for` si applica anche agli array: **vale per qualunque collezione, di qualunque tipo !**

JCF: INTERFACCE E IMPLEMENTAZIONI

- Per **usare** le collezioni, ovviamente ***non occorre conoscere l'implementazione***: basta attenersi alla specifica data dalle interfacce.
- Tuttavia, **scegliere una implementazione diventa necessario all'atto della costruzione** della collezione.
- .. ma non per pianificare il collaudo! 😊

Ora considereremo alcuni esercizi e li imposteremo *lasciando volutamente in bianco la fase di costruzione*, in modo da non legarci ad alcuna implementazione.

JCF: ALCUNI ESEMPI

Considereremo ora i seguenti esercizi:

- a) **Uso di Set** per operare su un **insieme** di elementi
 - esempio: un elenco di parole senza doppioni (Esercizio n.1)
- b) **Uso di List** per operare su una **sequenza** di elementi
 - scambiando due elementi nella sequenza (Esercizio n.2)
 - o iterando dal fondo con un iteratore di lista (Esercizio n.3)
- c) **Uso di Map** per **fare una tabella** di elementi (e contarli)
 - esempio: contare le occorrenze di parole (Esercizio n.4)
- d) **Uso di SortedMap** per creare un **elenco ordinato**
 - idem, ma creando poi un elenco ordinato (Esercizio n.5)
- e) **Uso dei metodi della classe Collections** per ordinare una collezione di oggetti (ad es. **Persone**)

ESERCIZIO 1 – Set

- Il problema: analizzare un **insieme** di parole
 - ad esempio, gli argomenti della riga di comando
 - e specificatamente:
 - stampare tutte le parole duplicate
 - stampare il numero di parole distinte
 - stampare la lista delle parole distinte
- A questo fine, usiamo un'istanza di **Set**
 - **non importa quale implementazione!**
 - e poi:
 - aggiungiamo ogni parola al Set tramite il metodo **add**: se è già presente, *non viene reinserita* e **add** restituisce *false*
 - alla fine stampiamo la dimensione (con **size**) e il contenuto (con **toString**) dell'insieme.

ESERCIZIO 1 – Set

```
import java.util.*;

public class FindDups {

    public static void main(String[] args) {

        Set s = new _____;

        for (int i=0; i<args.length; i++)
            if (!s.add(args[i]))
                System.out.println("Parola duplicata: " + args[i]);

        System.out.println(s.size() + " parole distinte: "+s);
    }
}
```

Un'implementazione qualsiasi di Set (la sceglieremo dopo!)


Output atteso:

```
>java FindDups Io sono Io esisto Io parlo
Parola duplicata: Io
Parola duplicata: Io
4 parole distinte: [Io, parlo, esisto, sono]
```

nessun ordine

ESEMPIO: Set CON ITERATORE

Per **elencare tutti gli elementi** di una collezione, ci si può anche ***procurare un iteratore per quella collezione***



```
...  
for (Iterator i = s.iterator(); i.hasNext(); ) {  
    System.out.print(i.next() + " ");  
}
```

Per ottenere un iteratore su una data collezione basta chiamare su di essa il metodo `iterator`.

È comunque preferibile usare il nuovo costrutto `for` (`foreach` in C#), in quanto più espressivo.

ESEMPIO: Set CON NUOVO FOR

```
...  
for (Object o : s) {  
    System.out.print(o + " ");  
}
```

ESERCIZIO 2 – List

- Il problema: **scambiare** due elementi in una *lista*
 - ad esempio, due parole in una lista di parole
 - più specificatamente:
 - ci serve una funzione accessoria (statica) **swap**
 - notare che *la nozione di scambio presuppone quella di posizione*, perché solo così si dà senso al termine "scambiare" (che si intende "scambiare di posizione")
- A questo fine, usiamo un'istanza di **List**
 - **non importa quale implementazione!**
 - e poi:
 - aggiungiamo ogni parola alla List tramite il metodo **add**
 - la stampiamo per vederla prima dello scambio
 - **effettuiamo lo scambio**
 - infine, la ristampiamo per vederla dopo lo scambio

ESERCIZIO 2 – List

La funzione di scambio:

```
static void swap(List a, int i, int j) {  
    Object tmp = a.get(i);  
    a.set(i, a.get(j)); a.set(j, tmp);  
}
```

Il main dell'esempio:

```
public static void main(String a  
  
    List list = new _____;  
    for (int i=0; i<args.length; i++) list.add(args[i]);  
    System.out.println(list);  
    swap(list, 2, 3);  
    System.out.println(list)  
}
```

Un'implementazione qualsiasi di
List (la sceglieremo dopo!)

Elementi n. 2 e 3
(3° e 4°) scambiati

```
java EsList cane gatto pappagallo  
canarino cane canarino pescerosso  
[cane, gatto, pappagallo, canarino,  
cane, canarino, pescerosso]  
[cane, gatto, canarino, pappagallo,  
cane, canarino, pescerosso]
```

Da Iterator A ListIterator

- In aggiunta al concetto generale di iteratore, comune a tutte le collezioni, **List** introduce il concetto specifico di *iteratore di lista* (*ListIterator*)
- Esso sfrutta le nozioni di *sequenza* e *posizione* peculiari delle liste per:
 - andare anche "a ritroso"
 - avere un concetto di "indice" e conseguentemente offrire metodi per tornare all' *indice precedente*, avanzare all'*indice successivo*, etc
- Perciò, è possibile anche ottenere un iteratore di lista *preconfigurato per iniziare da uno specifico indice*.

L'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  
}
```

La lista ha un concetto di posizione ed è navigabile anche a ritroso

Ergo, l'iteratore di lista ha i concetti di "prossimo indice" e "indice precedente"

ESERCIZIO 3

List & ListIterator

Si può ottenere un iteratore di lista che inizi da un indice specificato

Schema tipico di iterazione a ritroso:

```
for( ListIterator i = l.listIterator(l.size()) ;  
    i.hasPrevious() ; ) {  
    ...  
}
```

Per usare hasPrevious, occorre ovviamente iniziare dalla fine

Esempio: riscrittura a rovescio degli argomenti passati

```
public class EsListIt {  
    public static void main(String args[]) {  
        List l = new _____ ;  
        for (int i=0; i<args.length; i++) l.add(args[i]);  
        for( ListIterator i = l.listIterator(l.size()) ;  
            i.hasPrevious() ; )  
            System.out.print(i.previous()+" ");  
    }  
}
```

```
java EsListIt cane gatto cane canarino  
canarino cane gatto cane
```

ESERCIZIO 4 – Map

Obiettivo: conta le occorrenze delle parole digitate sulla linea di comando.

```
import java.util.*;
public class ContaFrequenza {
    public static void main(String args[]) {
        Map m = new _____;
        for (int i=0; i<args.length; i++) {
            Integer freq = (Integer) m.get(args[i]);
            m.put(args[i], (freq==null ? new Integer(1) :
                                   new Integer(freq.intValue() + 1)));
        }
        System.out.println(m.size() + " parole distinte:");
        System.out.println(m);
    }
}
```

put richiede un Object,
int non lo è → boxing

In realtà, oggi il boxing è
automatico → si può non
scriverlo in esplicito

```
>java ContaFrequenza cane gatto cane gatto gatto cane pesce
3 parole distinte: {cane=3, pesce=1, gatto=3}
```


ESERCIZIO 5 – SortedMap

Lo stesso esercizio con una *tabella ordinata*:

```
import java.util.*;

public class ContaFrequenzaOrd {

    public static void main(String args[]) {

        SortedMap m = new _____;

        for (int i=0; i<args.length; i++) {
            Integer freq = (Integer) m.get(args[i]);
            m.put(args[i], (freq==null ? new Integer(1) :
                                new Integer(freq.intValue() + 1)));
        }

        System.out.println(m.size()+" parole distinte:");
        System.out.println(m);
    }
}
```

```
>java ContaFrequenza cane gatto cane gatto gatto cane pesce
3 parole distinte: {cane=3, pesce=1, gatto=3}
```

```
>java ContaFrequenzaOrd cane gatto cane gatto gatto cane pesce
3 parole distinte: {cane=3, gatto=3, pesce=1}
```

33
elenco ordinato!

ESERCIZIO 6 – Collections

Come esempio d'uso dei metodi di **Collections** e della analoga classe **Arrays**, supponiamo di voler:

- costruire un array di elementi comparabili
 - ad esempio, un array di istanze di *Persona*, che supponiamo implementi l'interfaccia *Comparable*

- ottenerne una lista

`Arrays.asList(array)`

- ordinare tale lista

`Collections.sort(lista)`

OSSERVAZIONE: `Arrays.asList` restituisce un'istanza di "qualcosa" che implementa *List*, ma non si sa (e non serve sapere) esattamente cosa

UNA Persona COMPARABILE

```
class Persona implements Comparable {  
    private String nome, cognome;  
    public Persona(String nome, String cognome) {  
        this.nome = nome;  this.cognome = cognome;  
    }  
    public String nome() {return nome;}  
    public String cognome() {return cognome;}  
    public String toString() {return nome + " " + cognome;}  
  
    public int compareTo(Object x) {  
        Persona p = (Persona) x;  
        int confrontoCognomi = cognome.compareTo(p.cognome);  
        return (confrontoCognomi!=0 ? confrontoCognomi :  
                nome.compareTo(p.nome) );  
    }  
}
```

**Confronto lessicografico
fra stringhe**

**.. e se volessimo
ordinarle *in base al
cognome più lungo?***

ESERCIZIO 6: ordinamento di liste

```
class NameSort {  
    public static void main(String args[]) {  
        Persona elencoPersone[] = {  
            new Persona("Eugenio", "Bennato"),  
            new Persona("Roberto", "Benigni"),  
            new Persona("Edoardo", "Bennato"),  
            new Persona("Bruno", "Vespa")  
        };  
  
        List l = Arrays.asList(elencoPersone);  
        Collections.sort(l);  
        System.out.println(l);  
    }  
}
```

Produce una List (non si sa quale implementazione!) a partire dall'array dato

Ordina tale List in senso ascendente

Se il cognome è uguale, valuta il nome

```
>java NameSort  
[Roberto Benigni, Edoardo Bennato, Eugenio Bennato, Bruno Vespa]
```

JCF : dalle interfacce alle implementazioni

JCF: QUADRO GENERALE

		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Deque		ArrayDeque		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Implementazioni fondamentali:

- per Set: *HashSet, TreeSet, LinkedHashSet*
- per List: *ArrayList, LinkedList*
- per Map: *HashMap, TreeMap, LinkedHashMap*
- per Deque: *ArrayDeque, LinkedList*

In particolare, di queste adottano una *struttura ad albero TreeSet e TreeMap.*

QUALI IMPLEMENTAZIONI USARE?

		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Deque		ArrayDeque		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Regole generali per Set e Map:

- **se è indispensabile l'ordinamento, TreeMap e TreeSet**
(perché sono le uniche implementazioni di SortedMap e SortedSet)
- **altrimenti, preferire HashMap e HashSet perché molto più efficienti**
(tempo di esecuzione costante anziché $\log(N)$)

Regole generali per List:

- **di norma, meglio ArrayList**, che ha *tempo di accesso costante*
(anziché lineare con la posizione) essendo realizzata su array
- **preferire però LinkedList** se l'operazione più frequente è
l'aggiunta in testa o l'eliminazione di elementi in mezzo

RIPRENDENDO GLI ESEMPI...

Nell'esercizio n. 1 (Set) si può scegliere fra:

- **HashSet**: insieme non ordinato, tempo d'accesso costante
- **TreeSet**: insieme ordinato, tempo di accesso non costante

Output con HashSet:

```
>java FindDups Io sono Io esisto Io parlo  
Parola duplicata: Io  
Parola duplicata: Io  
4 parole distinte: [Io, parlo, esisto, sono]
```

ordine qualunque

Output con TreeSet:

```
>java FindDups Io sono Io esisto Io parlo  
Parola duplicata: Io  
Parola duplicata: Io  
4 parole distinte: [Io, esisto, parlo, sono]
```

ordine alfabetico!

RIPRENDENDO GLI ESEMPI...

Negli esercizi n. 2 (`List`) si può scegliere fra:

- **ArrayList**: i principali metodi eseguono in tempo costante, mentre gli altri eseguono in un tempo lineare, ma con una costante di proporzionalità molto più bassa di `LinkedList`.
- **LinkedList**: il tempo di esecuzione è quello di una tipica realizzazione basata su puntatori; implementa anche le interfacce `Queue` e `Deque`, offrendo così una coda FIFO

L'output però *non varia* al variare dell'implementazione, in ossequio sia al concetto di lista come *sequenza* di elementi, sia alla semantica di `add` come "append":

```
java EsList cane gatto pappagallo canarino cane canarino pescerosso  
[cane, gatto, pappagallo, canarino, cane, canarino, pescerosso]  
[cane, gatto, canarino, pappagallo, cane, canarino, pescerosso]
```

RIPRENDENDO GLI ESEMPI...

Nell'esercizio n. 4 (Map) si può scegliere fra:

- **HashMap**: tabella non ordinata, tempo d'accesso costante
- **TreeMap**: tabella ordinata, tempo di accesso non costante
- **LinkedHashMap**: tabella ordinata, tempo d'accesso costante ma con costante di proporzionalità più alta

Output con HashMap:

```
>java HashMapFreq cane gatto cane gatto gatto cane pesce  
3 parole distinte: {cane=3, pesce=1, gatto=3}
```

Output con TreeMap e LinkedHashMap (*elenco ordinato*):

```
>java TreeMapFreq cane gatto cane gatto gatto cane pesce  
3 parole distinte: {cane=3, gatto=3, pesce=1}
```

JCF "CLASSICA": LIMITI E PROBLEMI

- La JCF classica è stata usata per anni in molte applicazioni: ciò ne ha messo in luce pregi e *limiti*.
- In particolare, **l'uso del tipo `Object` come mezzo per ottenere genericità si è rivelato inadeguato**
 - all'epoca era una scelta inevitabile, l'unica per avere collezioni usabili "con qualunque tipo di oggetto"
 - ma ***equivale di fatto a disattivare il controllo di tipo***
 - di conseguenza, rende possibili ***operazioni sintatticamente corrette*** (che si compilano) ma ***semanticamente errate*** (che come tali danno poi errore a run-time)
- Questo ha portato a una riprogettazione globale della JCF alla luce del *nuovo concetto* di ***tipo generico***.