

Strutture Dati nella piattaforma Java: Java Collections Framework

STRUTTURE DATI IN JAVA

- **Java Collections Framework (JCF)** fornisce il supporto a qualunque tipo di *struttura dati*
 - **interfacce**
 - **una classe Collections** che definisce **algoritmi polimorfi** (funzioni statiche)
 - **classi** che forniscono **implementazioni** dei vari tipi di strutture dati specificati dalle interfacce
- **Obiettivo: strutture dati per "elementi generici"** (*vi ricordate il tipo degli elementi nell'ADT list e tree?*)

UN PO' DI DOCUMENTAZIONE IN RETE

Tutorial di Java sulle Collections:

<http://java.sun.com/docs/books/tutorial/collections/index.html>

Gerarchia delle interfacce (la vediamo subito):

<http://java.sun.com/docs/books/tutorial/collections/interfaces/index.html>

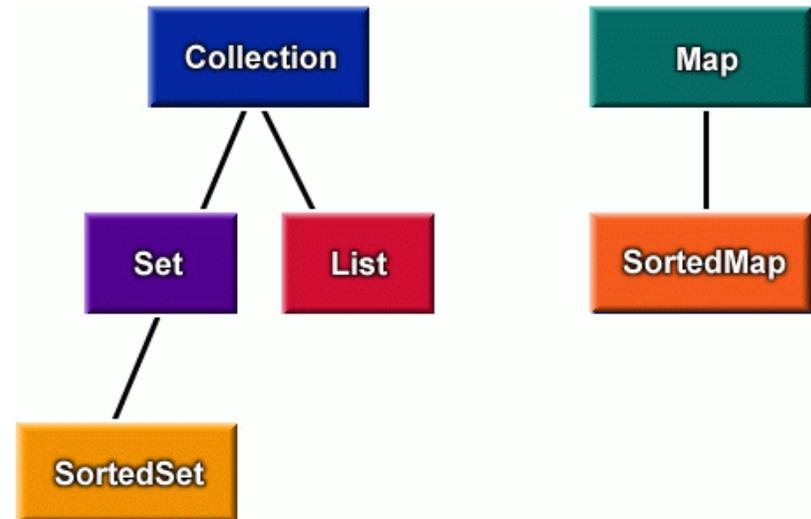
Classe Collections:

<http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html>

JAVA COLLECTIONS 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 associativa*, ossia tabella che associa chiavi a valori
- ***SortedMap***: una mappa (tabella) *ordinata*



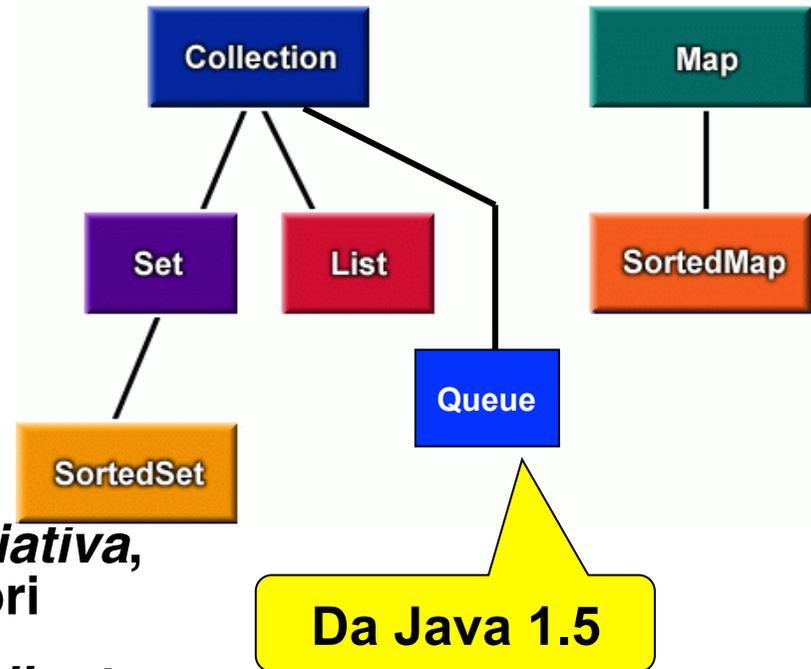
Criteri-guida per la definizione delle interfacce:

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

JAVA COLLECTIONS 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 associativa*, 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

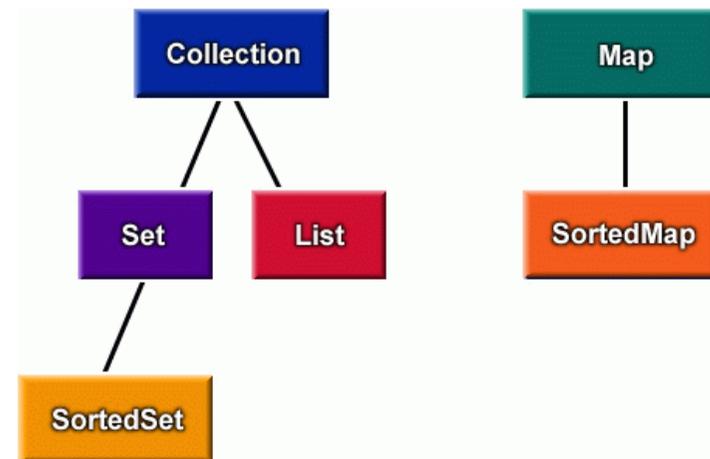
Collection introduce l'idea di *collezione* di elementi

- 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**
 - rimuovere un elemento dalla collezione **remove**
 - verificare se un elemento è nella collezione **contains**
 - 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**
 - ... e altri ... (si vedano le API Java)

L'INTERFACCIA Set

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

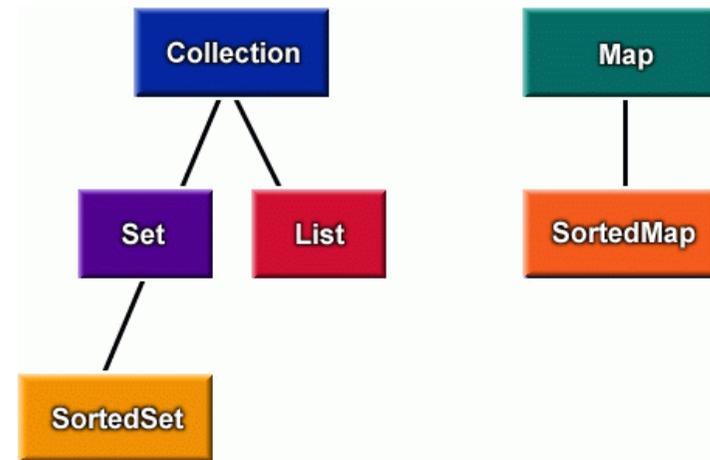
- in quanto insieme, *non ammette elementi duplicati e non ha una nozione di sequenza o di posizione*
- *l'interfaccia di accesso* non cambia sintatticamente, ma *si aggiungono vincoli al contratto d'uso*:
 - **add** aggiunge un elemento solo se esso non è già presente
 - **equals** assicura che due 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

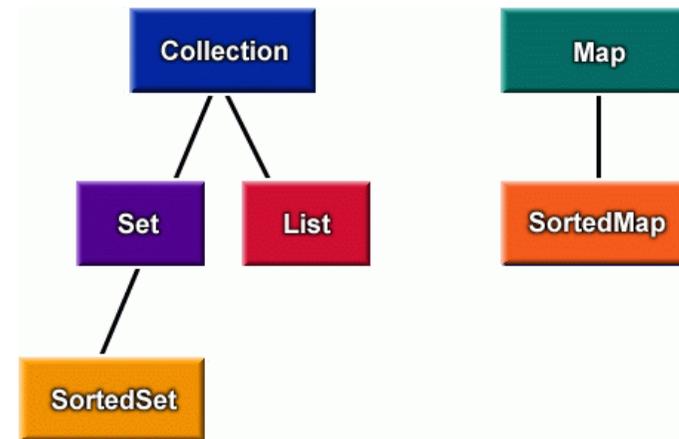
- *tipicamente ammette duplicati*
- in quanto sequenza, ha una nozione di *posizione*
- *l'interfaccia di accesso* aggiunge sia *vincoli al contratto d'uso*, sia *nuovi metodi per l'accesso posizionale*
 - `add` aggiunge un elemento in fondo alla lista (append)
 - `equals` è vero se gli elementi corrispondenti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

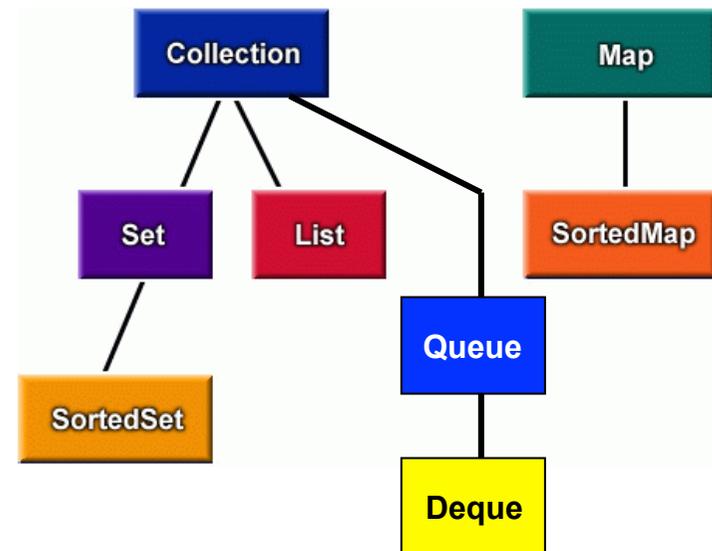


LE INTERFACCE Queue E Deque

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** (in cui si possono inserire/togliere elementi da ambo le estremità)

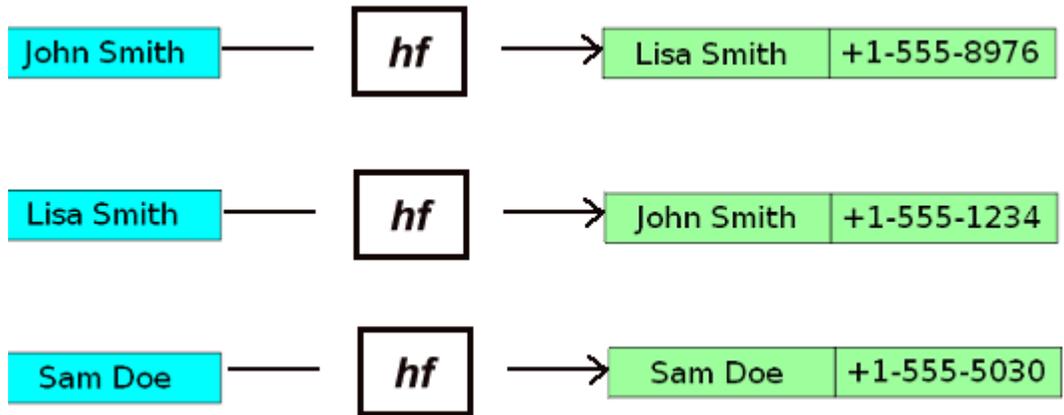


L'INTERFACCIA Map

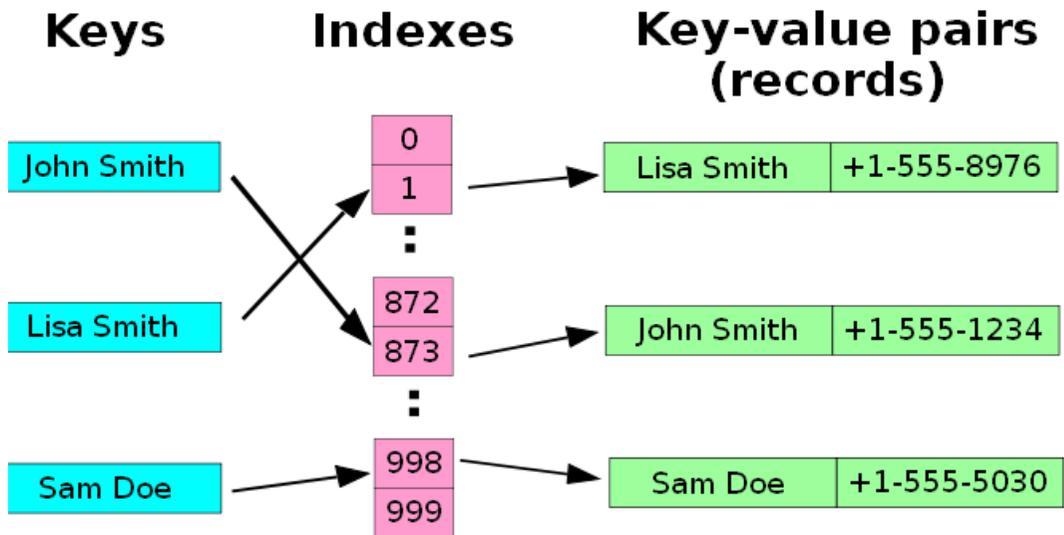
Map introduce l'idea di *tabella di elementi*, ognuno *associato univocamente a una chiave identificativa*.

- in pratica, è una *tabella a due colonne (chiavi, elementi)* in cui i dati della prima colonna (*chiavi*) identificano univocamente la riga.
- l'idea di fondo è *riuscire ad accedere "rapidamente" a ogni elemento, data la 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 INDICI per guidare il reperimento dell'elemento a partire dalla chiave.

L'INTERFACCIA Map



Mappe basate su *funzioni hash*



Mappe basate su *indici*

L'INTERFACCIA Map

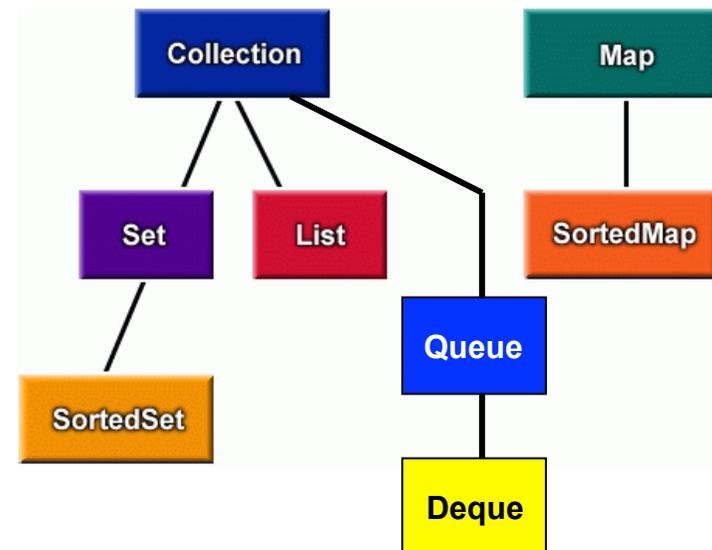
- **L'interfaccia di accesso** prevede metodi per :
 - inserire in tabella una coppia (*chiave, elemento*) **put**
 - accedere a un elemento in tabella, data la chiave **get**
 - verificare se una *chiave* è presente in tabella **containsKey**
 - verificare se un *elemento* è presente in tabella **containsValue**
- inoltre, **supporta le cosiddette "Collection views"**, ovvero metodi per recuperare *insiemi significativi*:
 - l'insieme di tutte le chiavi **keySet**
 - la collezione di tutti gli elementi **values**
 - l'insieme di tutte le righe, ovvero delle coppie (*chiave, elemento*) **entrySet**

L'INTERFACCIA SortedMap

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

- l'ordinamento è **quello naturale degli elementi** (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 tabella 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`).
 - Altrimenti, il metodo farà una ricerca binaria basata su iteratore, che effettua $O(n)$ attraversamenti di link e $O(\log n)$ confronti.

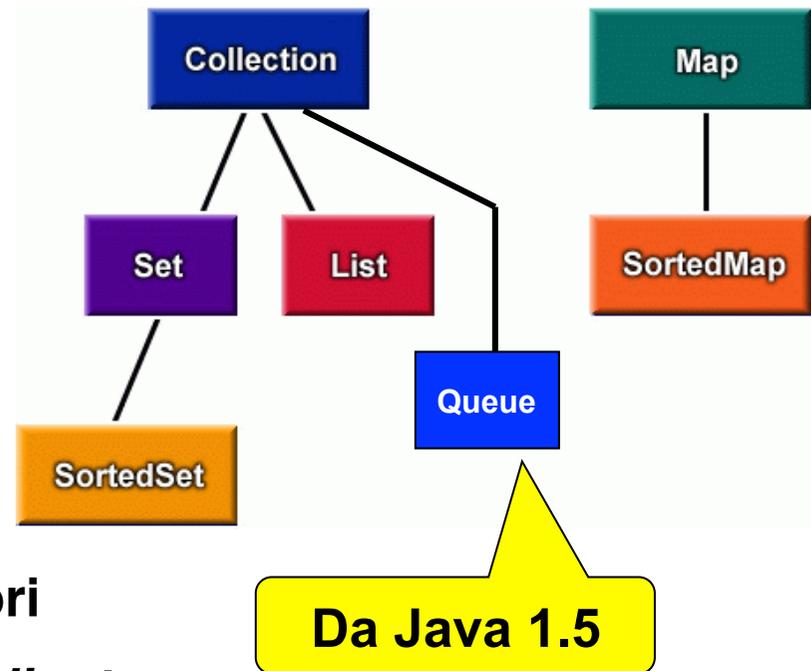
JCF – recap (Lezione in laboratorio)

- **Java Collection Framework (JCF)** fornisce il supporto a qualunque tipo di *struttura dati*
 - **interfacce**
 - **una classe Collections** che definisce **algoritmi polimorfi** (funzioni statiche)
 - **classi** che forniscono **implementazioni** dei vari tipi di strutture dati specificati dalle interfacce
- **Obiettivo: strutture dati per "elementi generici"** (*vi ricordate il tipo degli elementi nell'ADT list e tree?*)

JAVA COLLECTIONS 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!)

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 (new)*** della collezione

JCF: QUADRO GENERALE (un po' datato ...)

		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, Vector*
- per Map: *HashMap, TreeMap, Hashtable*
- per Deque: *ArrayDeque, LinkedList*

In particolare, di queste adottano una *struttura ad albero* *TreeSet e TreeMap* (implementano *SortedSet* e *SortedMap*)

Classi pre-JCF, poi reingegnerizzate in accordo alle interfacce `List` e `Map`

**Qualche esempio
(ed esercizio)**

JCF: ALCUNI ESEMPI

- a) **Uso di Set** per operare su un **insieme** di elementi
 - esempio: un elenco di parole senza doppioni (Esercizio n.1)

- b) **Uso di ArrayList** per operare su una sequenza di elementi
 - **Iterando** sulla 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)

- e) **Uso di SortedMap** per creare un elenco ordinato
 - idem, ma creando poi un elenco ordinato (Esercizio n.5)

- f) **Uso dei metodi della classe Collections** per ordinare una collezione di oggetti (ad es. Persone)

ESERCIZIO n° 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**
 - **Variamo l'implementazione (HashSet, poi ...)**
- **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 n° 1 – Set

```
import java.util.*;

public class FindDups {

    public static void main(String args[]){

        // dichiarazione dell'oggetto s di tipo Set
        // e sua creazione come istanza di HashSet
        // ciclo per aggiungere ad s (s.add) ogni argomento
        // se non inserito, stampa "Parola duplicata"
        // stampa cardinalità (s.size) e l'insieme s
    }
}
```

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

ESERCIZIO n° 1 – Set

```
import java.util.*;

public class FindDups {
    public static void main(String args[]) {
        Set s = new HashSet();
        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);
    }
}
```

Con una diversa implementazione?
Ad esempio,
TreeSet

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

INTERFACCE E IMPLEMENTAZIONI

		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

ESERCIZIO n° 1 – Set

```
import java.util.*;
public class FindDups {
    public static void main(String args[]){
        Set s = new TreeSet();
        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);
    }
}
```

IMPLEMENTAZIONE ES. n° 1 – Set

Nell'esercizio n. 1 (Set) **in fase di costruzione della collezione** si può scegliere una diversa implementazione, ad esempio, 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: [esisto, Io, parlo, sono]
```

ordine alfabetico!

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

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 essa l'apposito metodo `iterator`**

ESERCIZIO n° 2 – ArrayList

- Il problema: creiamo una **lista** di parole
 - ad esempio, gli argomenti della riga di comando
- e poi:
 - Stampiamole una per linea, utilizzando un iteratore

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

- A questo fine, usiamo un'istanza di **List**
 - **Definiamo l'implementazione (ad es., ArrayList)**
- e poi:
 - Creiamo l'iteratore
 - e stampiamo ciascuna parola

ESERCIZIO n° 2 – List

```
import java.util.*;
public class IteraStampa {
    public static void main(String args[]) {
        List al = new ArrayList();
        for (int i=0; i<args.length; i++)
            al.add(args[i]);
        Iterator itr = al.iterator();
        while(itr.hasNext())
        { String element = (String)itr.next();
          System.out.print("Parola: "+ element);
          System.out.println();
        }
    }
}
```

```
>java IteraStampa Io sono Io esisto
Parola: Io
Parola: sono
Parola: Io
Parola: esisto
```

ESERCIZIO n° 2 – List

```
import java.util.*;
public class IteraStampa {
    public static void main(String args[]){
        List al = new ArrayList();
        for (int i=0; i<args.length; i++)
            al.add(args[i]);
        for( Iterator itr = al.iterator();itr.hasNext(); )
        { String element = (String) itr.next();
          System.out.print("Parola: "+ element);
          System.out.println();
        }
    }
}
```

```
>java IteraStampa Io sono Io esisto
Parola: Io
Parola: sono
Parola: Io
Parola: esisto
```

ITERATORI e NUOVO COSTRUTTO `for`

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

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

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

Dunque, il nuovo `for` non si applica solo agli array:
ma *vale per qualunque collezione, di qualunque tipo*

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"

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

```
ListIterator listIterator();  
ListIterator listIterator(int index);
```

ESERCIZIO n° 3

List & ListIterator

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 ArrayList();  
        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 n° 3 – List

```
import java.util.*;
public class IteraStampa {
    public static void main(String args[]) {
        List al = new ArrayList();
        for (int i=0; i<args.length; i++)
            al.add(args[i]);
        for( ListIterator itr = al.listIterator(al.size());
            itr.hasPrevious(); )
        { String element = (String) itr.previous();
          System.out.print("Parola: "+ element);
          System.out.println();
        }
    }
}
```

```
>java IteraStampa Io sono Io esisto
Parola: esisto
Parola: Io
Parola: sono
Parola: Io
```

ESERCIZIO n° 4 – Map

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

Utilizziamo come struttura dati di appoggio una tabella o **mappa associativa** (Map), che ad ogni parola (argomento della linea di comando) associa la sua frequenza

parola *frequenza*

Man mano che processiamo un argomento della linea di comando, aggiorniamo la frequenza di questa parola, accedendo alla tabella

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

TAVOLA E SUE REALIZZAZIONI

cane	3
gatto	3
pesce	1

Tempo richiesto dall'operazione più costosa (cercare l'elemento data la chiave, che in questo esempio è il campo *parola*):

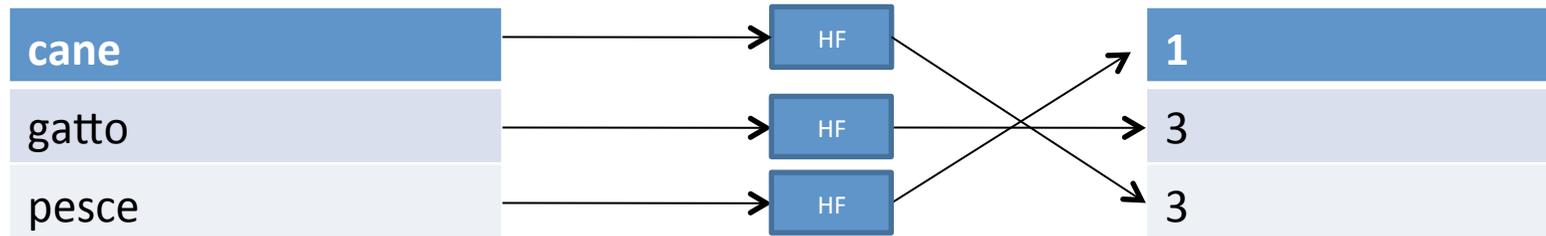
- Liste $O(n)$
- Alberi binari di ricerca non bilanciati $O(n)$
- Alberi binari di ricerca bilanciati $O(\log_2 n)$
- **Tabelle hash** 1

Tabelle ad accesso diretto

- Sono implementazioni di dizionari (tavole) basati sulla proprietà di accesso diretto alle celle di un array
- Idea:
 - dizionario memorizzato in array V di m celle
 - a ciascun elemento è associata una chiave intera nell'intervallo $[0, m-1]$
 - elemento con chiave k contenuto in $V[k]$
 - al più $n \leq m$ elementi nel dizionario

Tabelle hash (*hash map*)

- Fa corrispondere una data *chiave* con un dato *valore* (*indice*) attraverso una **funzione hash**



- Usate per l'implementazione di strutture dati associative astratte come **Map** o **Set**

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**
- inoltre, **supporta le cosiddette "Collection views"**, ovvero metodi per recuperare *insiemi significativi*:
 - l'insieme di tutte le chiavi **keySet**
 - la collezione di tutti gli elementi **values**
 - l'insieme di tutte le righe, ovvero delle coppie (*chiave, elemento*) **entrySet**

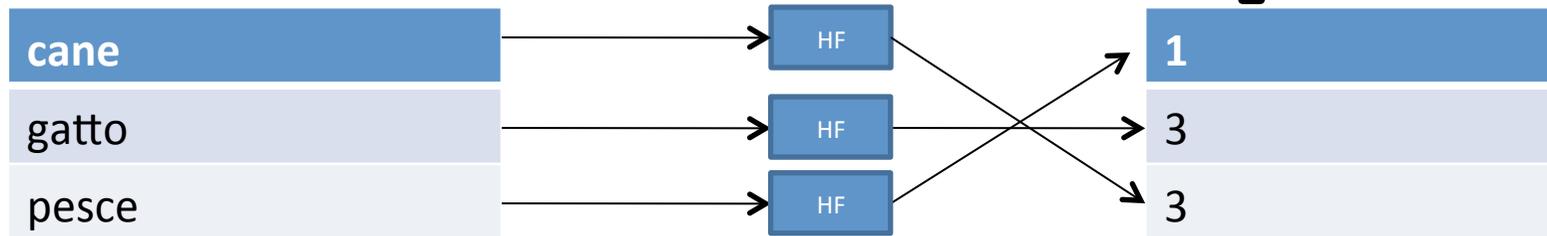
ESERCIZIO n° 4 – Map

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

```
import java.util.*;
public class ContaFrequenza {
    public static void main(String args[]) {
        // dichiarazione dell'oggetto m di tipo Map
        // e sua creazione come istanza di _____
        // per ciascun argomento della linea di comando
            // preleva (get) da m la sua frequenza
            // aggiorna (put) la coppia <arg,freq> in m
            // con frequenza incrementata di 1
        // stampa cardinalità (m.size) e la tabella m
    }
```

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

ESERCIZIO n° 4 – Map



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

put richiede un Object, int non lo è → boxing
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}
```

OBIETTIVO: GENERICITÀ

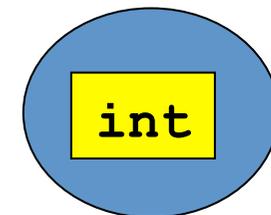
- **Nella JCF "classica"** (\leq JDK 1.4) il mezzo per ottenere **collezioni di elementi generici** è stata l'adozione del **tipo "generico" *Object*** come **tipo dell'elemento**
 - i metodi che aggiungono / tolgono **oggetti** dalle collezioni prevedono un **parametro di tipo *Object***
 - i metodi che cercano / restituiscono **oggetti** dalle collezioni prevedono un **valore di ritorno *Object***
 - rischi di **disuniformità** negli oggetti contenuti
 - problemi di **correttezza nel type system** (*downcasting*)

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 !
- **SOLUZIONE**: *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>



Integer

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

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

ESERCIZIO n° 4 – Map

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

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

put richiede un Object, int non lo è → boxing
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}
```

IMPLEMENTAZIONE ...

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

Output con HashMap:

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

Output con TreeMap (*elenco ordinato*):

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

ESERCIZIO n° 5 – SortedMap

Lo stesso esercizio con una *tabella ordinata*:

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

E' possibile solo
TreeMap()

```
>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}
```

elenco ordinato!

OBIETTIVO: GENERICITÀ

- **Nella JCF "classica"** (\leq JDK 1.4) il mezzo per ottenere **collezioni di elementi generici** è stata l'adozione del **tipo "generico" *Object*** come **tipo dell'elemento**
 - i metodi che aggiungono / tolgono *oggetti* dalle collezioni prevedono un parametro di tipo *Object*
 - i metodi che cercano / restituiscono *oggetti* dalle collezioni prevedono un valore di ritorno *Object*
 - rischi di *disuniformità* negli oggetti contenuti
 - problemi di *correttezza* nel type system (*downcasting*)
- **La successiva JCF "generica"** (\geq JDK 1.5) si basa perciò sul nuovo concetto di **tipo generico (trattati nel corso di Ingegneria del Software)**

ESEMPIO n° 4 bis– Map

Creazione di una tabella di numeri, dove il nome del numero è la chiave. Usiamo i tipi per specializzare la tabella generica:

```
import java.util.*;
public class TabellaNumeri {
    public static void main(String args[]) {
        Map<String, Integer> numbers =
            new Hashtable<String, Integer>();
        numbers.put("one", 1);
        numbers.put("two", 2);
        numbers.put("three", 3);
        // ricerca di un numero, in base al nome
        Integer n = numbers.get("two");
        if (n != null)
            { System.out.println("two = " + n);}
        else {System.out.println("nessun numero"); }
    }
}
```

ESERCIZIO n° 4 ter – Map

L'esercizio 4 con Map, rivisto ...cambia poco

```
import java.util.*;
public class ContaFrequenza {
    public static void main(String args[]) {
        Map<String, Integer> m =
            new Hashtable<String, Integer>();
        for (int i=0; i<args.length; i++) {
            Integer freq = (Integer)m.get(args[i]);
            if (freq!=null) m.put(args[i], freq + 1);
            else m.put(args[i],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}
```

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 tabella 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`).
 - Altrimenti, il metodo farà una ricerca (binaria) basata su iteratore, che effettua $O(n)$ attraversamenti di link e $O(\log n)$ confronti.

ESERCIZIO n° 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 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 occorre 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**

ESERCIZIO n° 6: ordinamento di liste

```
public class NameSort {  
    public static void main(String args[]) {  
        // definizione e creazione di un array di Persone  
        // dall'array con il metodo statico Arrays.asList  
        // ottieni una lista l del tipo List  
        // ordina l con il metodo statico  
        Collections.sort    // stampa l  
    }  
}
```

Se il cognome è uguale, valuta il nome

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

ESERCIZIO n° 6: ordinamento di liste

```
public 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")  
        };  
    }  
}
```

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

```
List l = Arrays.asList(elencoPersone);
```

```
Collections.sort(l);
```

Ordina tale List in senso ascendente

```
System.out.println(l);
```

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, Vector*
- per Map: *HashMap, TreeMap, Hashtable*
- per Deque: *ArrayDeque, LinkedList*

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

Classi pre-JCF, poi reingegnerizzate in accordo alle interfacce List e Map

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