Strutture Dati nella piattaforma Java: Java Collections Framework

Collezioni

- Una **collezione** (o **contenitore**) consente di organizzare e gestire un gruppo di oggetti
  - collezioni (vere e proprie)
  - mappe (tavole chiave-attributi)
  - implementate nel **Java Collections Framework** (**JFC**)
    - un insieme di interfacce e classi
    - package **java.util**

STRUTTURE DATI IN JAVA

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

UN PO' DI DOCUMENTAZIONE IN RETE

The Collections Framework:

https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html

Documentazione Sun:

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

(https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html)

Gerarchia delle interfacce (la vediamo subito):

http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html

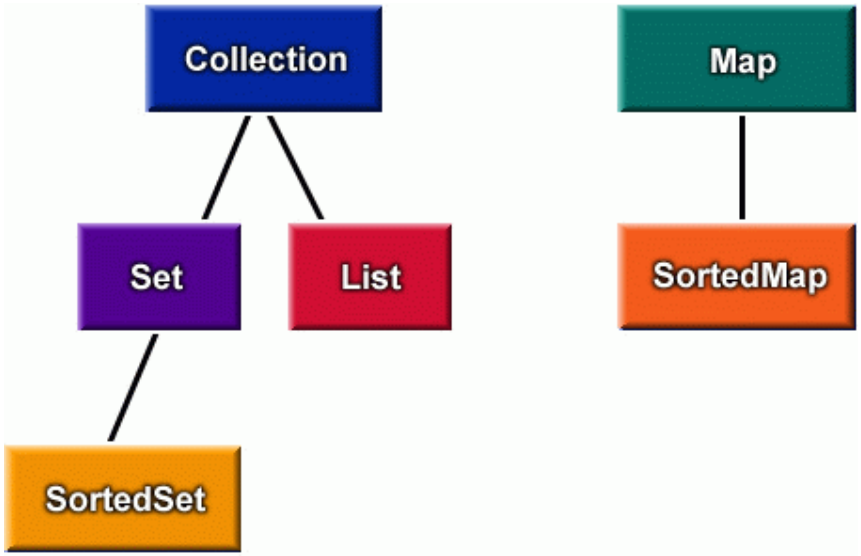
Classe Collections:

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

JCF: Interfacce (1)

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*

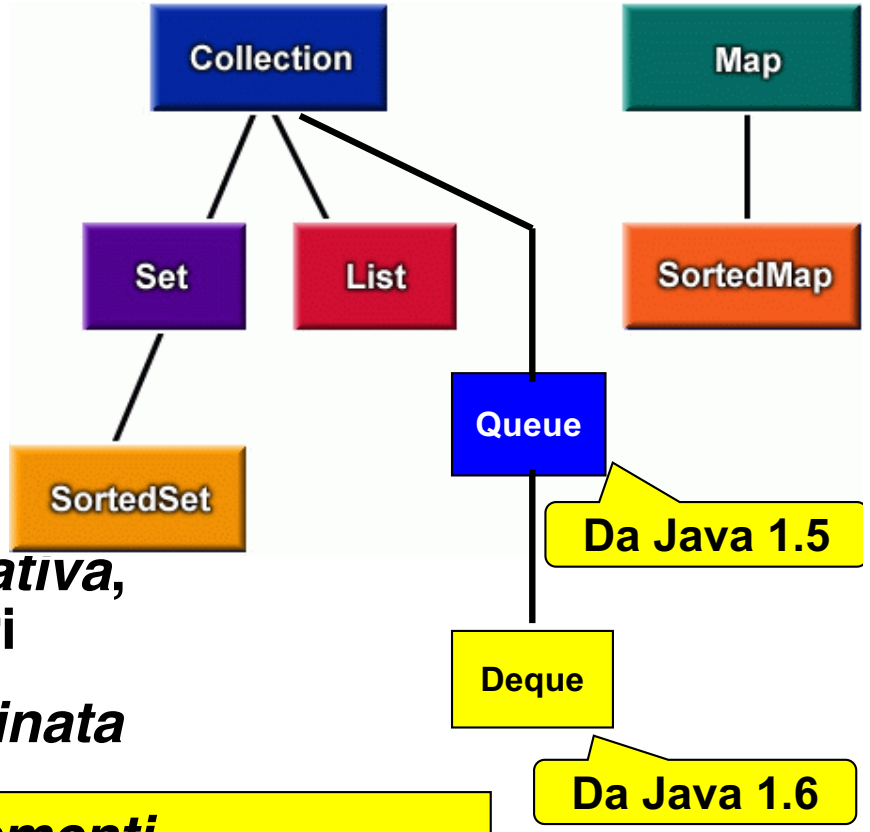
Criteri-guida per la definizione delle interfacce:

- **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!)*
- ***`Deque`***: inserzione e rimozione da entrambi i capi della lista

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 :**
  - **aggiungere un elemento 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)***

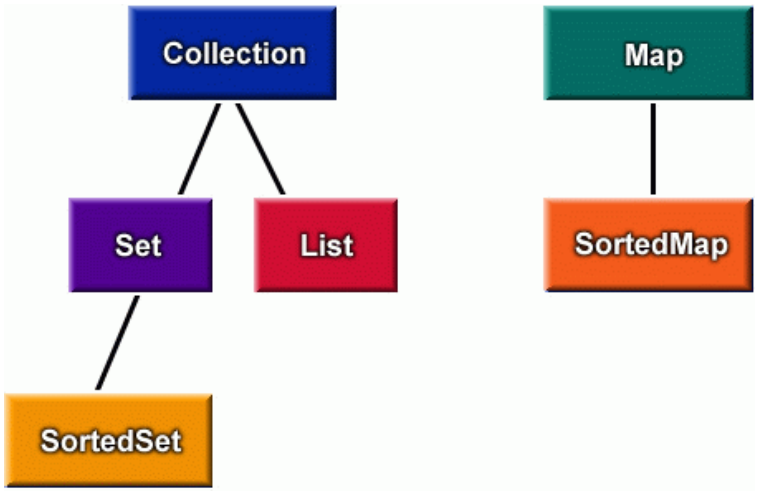
L'INTERFACCIA `Collection`

- Operazioni principali:
  - **`boolean add(Object o)`**
  - **`boolean contains(Object o)`**
  - **`boolean remove(Object o)`**
  - **`int size()`**
  - **`boolean isEmpty()`**
  - **`Iterator iterator()`**

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 `Set`

- Operazioni principali:
  - **`boolean add(Object o)`**
  - **`boolean contains(Object o)`**
  - **`boolean remove(Object o)`**
  - **`int size()`**
  - **`boolean isEmpty()`**
  - **`Iterator iterator()`**

Esempio: uso di un insieme di stringhe

```
import java.util.*; ...

Set s;                  // dichiara un insieme di String

s = new HashSet(); // crea un insieme s

s.add("alfa");     //inserisce tre oggetti String in s
s.add("beta");
s.add("gamma");

System.out.println( s.size() ); // 3
System.out.println( s.contains("alfa") ); // true
System.out.println( s.contains("sei") ); // false
System.out.println( s.toString() );
                        // [alfa, beta, gamma]
```

```
s.add("alfa");                          // s non cambia

System.out.println( s.size() );         // 3

s.remove("beta");
System.out.println( s.size() );         // 2
System.out.println( s.toString() );
                                        // [alfa, gamma]

s.add("delta");
s.add("epsilon");
System.out.println( s.size() );         // 4
```

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 contatore 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 "iterare" in una collezione con una modalità chiara e ben definita**

ITERATORI

Un *iteratore* è una entità capace di *garantire l'attraversamento di una collezione* con una modalità chiara e ben definita *(anche se la collezione venisse modificata)*

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

Di fatto, un iteratore offre un metodo **`next`** per esplorare uno ad uno gli elementi della collezione: il metodo **`hasNext`** permette di sapere quando non ci sono più elementi.

Per ottenere un iteratore per una data collezione, basta chiamare su essa l'apposito metodo `iterator()`

Iteratori

- Operazioni principali della classe `Iterator`:
  - **`boolean hasNext()`**
  - **`Object next()`**
  - **`void remove()`**

Esempio: uso di un iteratore

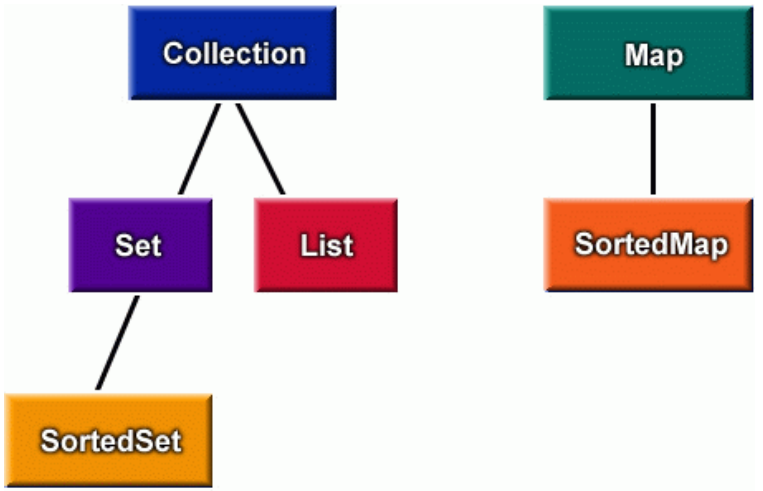
```
/* Visualizza gli elementi dell'insieme s */

public static void visualizza(Set s) {
// precondizione: s!=null

Iterator i;          // per visitare gli elementi di s
Object o;            // un elemento di s

/* visita e visualizza gli elementi di s */
i = s.iterator();          //creo iterator i
while ( i.hasNext() ) {
  o = i.next();
  System.out.println(o.toString()); }
}
```

```
/* Visualizza gli elementi dell'insieme s */

public static void visualizza(Set s) {
// precondizione: s!=null

Iterator i;          // per visitare gli elementi di s
String o;            // un elemento di s

/* visita e visualizza gli elementi di s */
i = s.iterator();          //creo iterator i
while ( i.hasNext() ) {
  o = (String) i.next();
  System.out.println(o); }
}
```

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*



Interfaccia `List`

- Operazioni principali:
  - `boolean add(Object o)`
  - `boolean contains(Object o)`
  - `boolean remove(Object o)`
  - `int size()`
  - `boolean isEmpty()`
  - `Iterator iterator()`

- Operazioni principali per l'accesso posizionale:
  - `Object get(int i)`
  - `Object set(int i, Object o)`
  - `Object remove(int i)`
  - `void add(int i, Object o)`
  - `int indexOf(Object o)`

Esempio: lista di String

```
List l;      // una lista di stringhe

/* crea una nuova lista l */
l = new ArrayList();

/* inserisce alcune stringhe nella lista l */
l.add("due");
l.add("quattro");
l.add("sei");

/* accede alla lista */
System.out.println( l.size() ); // 3
System.out.println( l.get(0) ); // due
System.out.println( l.toString() );
                              // [due, quattro, sei]
```

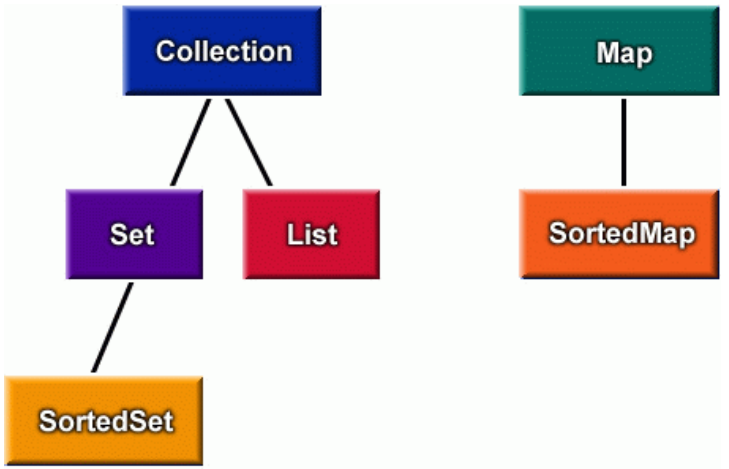
```
/* modifica la lista l */

l.add(1, "tre");     // tra "due" e "quattro"
System.out.println( l.toString() );
     // [due, tre, quattro, sei]
l.add(0, "uno");     // inserimento in testa

System.out.println( l.toString() );
          // [uno, due, tre, quattro, sei]
l.remove(4);                  // cancella "sei"
System.out.println( l.size() );    // 4
System.out.println( l.toString() );
          // [uno, due, tre, quattro]
```

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.

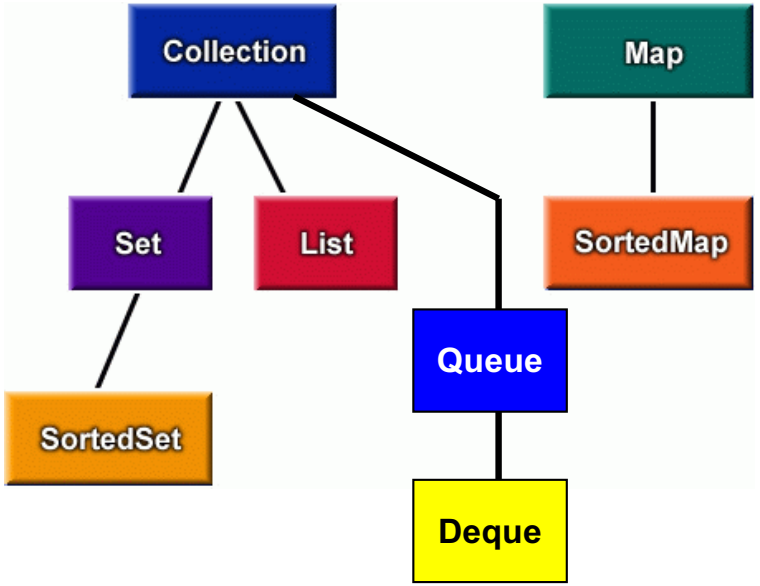


LE INTERFACCE `Queue` E `Deque`

`Queue` (≥ 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` (≥ JDK 1.6) specializza `Queue` con l'idea di *doppia coda*
(in cui si possono inserire/togliere elementi da ambo le estremità)

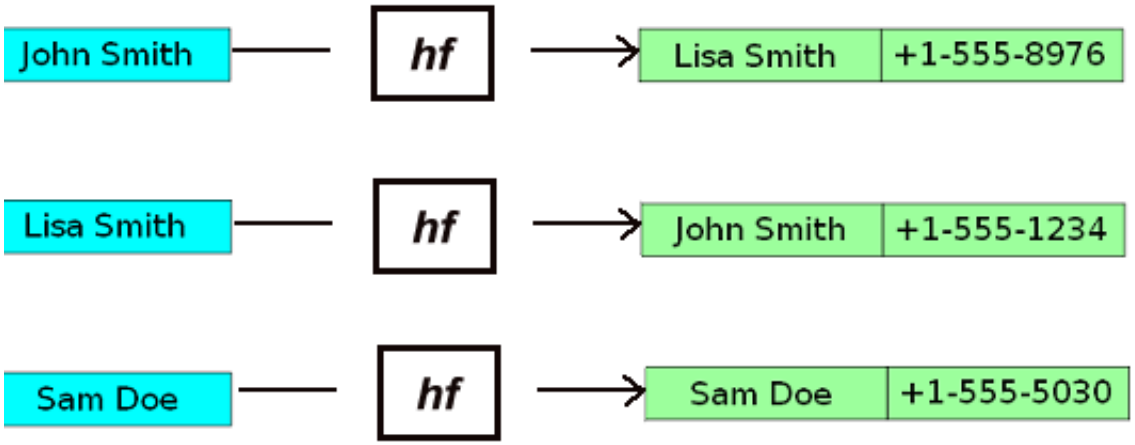


L'INTERFACCIA `Map`

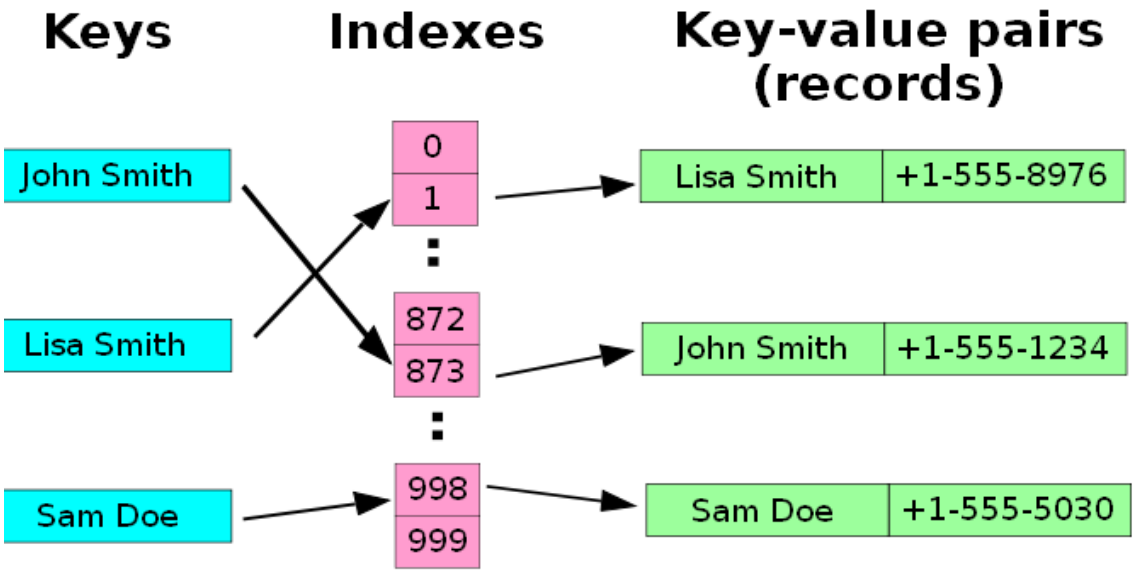
`Map` introduce l'idea di *tabella di elementi,* ognuno *associato univocamente a una chiave identificativa.*

- **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`

Interfaccia `Map`

Operazioni principali:

- `Object put(Object k, Object v)`
- `Object get(Object k)`
- `Object remove(Object k)`
- `int size()`
- `boolean isEmpty()`
- `Set keySet()`

Esempio: dizionario italiano-inglese

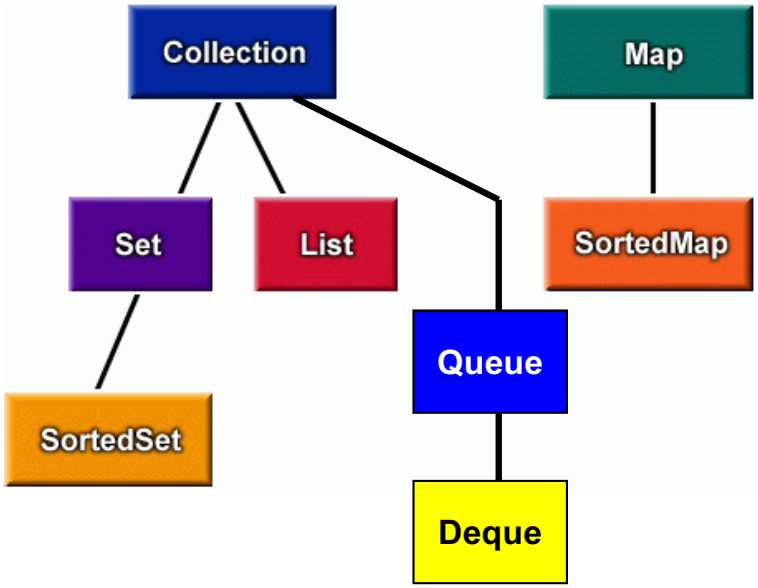
```
Map m;  // un insieme di coppie
                // (parola italiana, traduzione inglese)

/* crea una nuova mappa m */
m = new HashMap();

/* inserisce alcune coppie nella mappa m */
m.put("uno", "one");
m.put("due", "two");
m.put("tre", "tree");  // ooops ...

/* accede alla mappa */
System.out.println( m.size() );                 // 3
System.out.println( m.get("uno") );   // one
System.out.println( m.get("sei") );   // null System.out.println(
m.toString() );
                // {tre=tree, uno=one, due=two}
/* modifica la mappa m */
m.put("tre", "three");                  // corretto
System.out.println( m.size() );         // ancora 3 System.out.println(
m.get("tre") );                         // three
```

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.**



JCF: Classi

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

		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)
   - Iterando sull'insieme(Esercizio n. 1bis)

b) Uso di `List` 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)
   - scambiare due elementi nella lista (Esercizio 2bis)

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 il metodo `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(args[i])) 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

s.toString()

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 implentazione, 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 contatore 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

Un *iteratore* è una entità capace di *garantire l'attraversamento di una collezione* con una semantica chiara e ben definita *(anche se la collezione venisse modificata)*

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

Di fatto, un iteratore offre un metodo **`next`** per esplorare uno ad uno gli elementi della collezione: il metodo **`hasNext`** permette di sapere quando non ci sono più elementi.

Per ottenere un iteratore per una data collezione, basta chiamare su essa l'apposito metodo `iterator()`

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`

ITERATORI e NUOVO COSTRUTTO `for`

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

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

equivale a:

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

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

ESEMPIO: `Set` CON ITERATORE

Per elencare tutti gli elementi di una collezione, creiamo *un iteratore per quella collezione*

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

`TO DO .. .`

Output atteso:

```
>java FindDups Io sono Io esisto Io parlo
Io parlo esisto sono
```

ESEMPIO: `Set` CON ITERATORE

Per elencare tutti gli elementi di una collezione, creiamo *un iteratore per quella collezione*

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

Output atteso:

```
>java FindDups Io sono Io esisto Io parlo
Io parlo esisto sono
```

ESEMPIO: `Set` CON ITERATORE

Per elencare tutti gli elementi di una collezione, creiamo *un iteratore per quella collezione*

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

Output atteso:

```
>java FindDups Io sono Io esisto Io parlo
Io parlo esisto sono
```

ITERATORI e NUOVO COSTRUTTO `for`

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

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

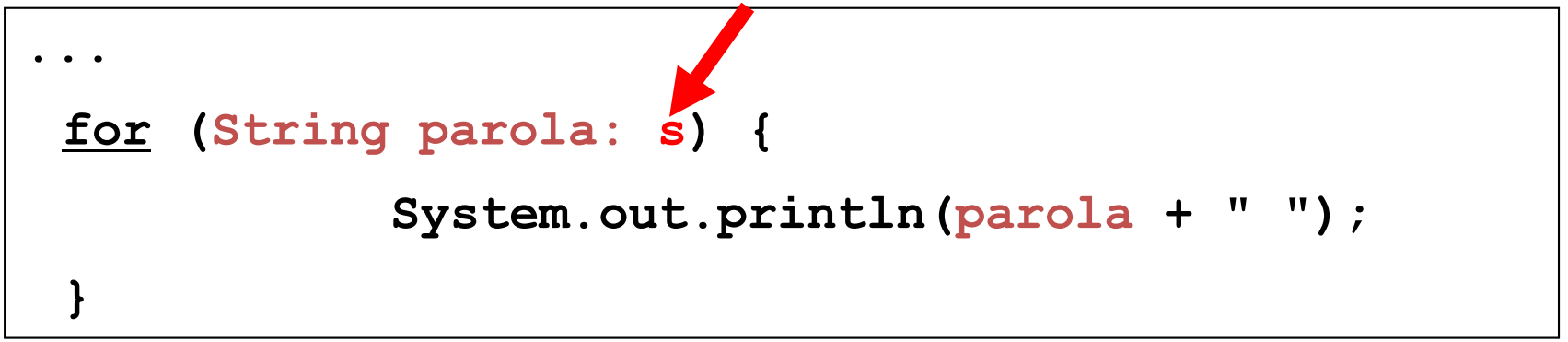
equivale a:

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

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

ESEMPIO `for`

Per elencare tutti gli elementi di una collezione, creiamo *un iteratore per quella collezione*



```
...
 for (String parola: s) {
            System.out.println(parola + " ");
 }
```

`for(x : coll)`

Output atteso:

```
>java FindDups Io sono Io esisto Io parlo
Io parlo esisto sono
```

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

- **A questo fine, usiamo un'istanza di `List`**
  - **Definiamo l'implementazione (ad es., ArrayList)**
- **e poi:**
  - **Creiamo l'Iterator**
  - **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 – 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 itr = al.iterator(); itr.hasNext(); )
        {/* operazioni su x = itr.next() */}
```

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

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
```

ESERCIZIO n° 2 – ArrayList

```
import java.util.*;
public class IteratorDemo { public static void
main(String args[]) {
// Create an array list
ArrayList al = new ArrayList();
// add elements to the array list
al.add("C"); al.add("A"); al.add("E");
al.add("B"); al.add("D"); al.add("F");
// Use iterator to display contents of al
System.out.print("Original contents of al: ");
Iterator itr = al.iterator(); while(itr.hasNext())
   { Object element = itr.next();
System.out.print(element + " "); }
System.out.println();
}
```

ITERATORI e NUOVO COSTRUTTO `for`

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

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

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

- *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.

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

Esempio enhanced `for`

Lavora su *collezioni* e *array*

```
class EnhancedForDemo
{ public static void main(String[] args)
  { int[] numbers = {1,2,3,4,5,6,7,8,9,10};
    for (int item : numbers)
      { System.out.println("Vale: " + item); }
  }
}
```

Vale: 1

Vale: 2

Vale: 3

etc etc

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(Object element1: al )
   { String element = (String) element1;
     System.out.print("Parola: "+ element);
     System.out.println();
     }
}
```

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

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"

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

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

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((String) 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° 3 – ArrayList

```
import java.util.*;
public class IteratorDemo { public static void
main(String args[]) {
// Create an array list
ArrayList al = new ArrayList();
// add elements to the array list
al.add("C"); al.add("A"); al.add("E");
al.add("B"); al.add("D"); al.add("F");
// Use iterator to display contents of al
System.out.print("Original contents of al: ");
ListIterator litr = al.listIterator();
while(litr.hasNext())
   {  Object element = litr.next();
      System.out.print(element + " ");
      System.out.println();
```

Usiamo `ListIterator`

```
// Now, display the list backwards
System.out.print(”Show list backwards: ");

while(litr.hasPrevious())
{ Object element = litr.previous();
  System.out.print(element + " "); }
  System.out.println(); }
}
```

L’iteratore è alla fine della lista, dal ciclo precedente

```
// Modify objects being iterated
ListIterator litr = al.listIterator();
while(litr.hasNext())
 { Object element = litr.next();
   litr.set(element + "+"); }
System.out.print("Modified contents of al: ");
itr = al.iterator();
while(itr.hasNext()) { Object element =
itr.next(); System.out.print(element + " "); }
System.out.println(); // Now, display the list
backwards System.out.print("Modified list
backwards: "); while(litr.hasPrevious()) {
Object element = litr.previous();
System.out.print(element + " "); }
System.out.println(); } }
```

ESERCIZIO n° 2bis – `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`**
  - **in costruzione, scegliamo quale implementazione (la varieremo)**
- **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 n° 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 args[]){
  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` (possiamo variarla)

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

Elementi n. 2 e 3 (3° e 4°) scambiati

IMPLEMENTAZIONE ES. n° 2 – `List`

Nell'esercizio 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**
- **`Vector`: versione reingegnerizzata e sincronizzata di ArrayList**

L'output però *non varia* al variare dell'implementazione, in accordo 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]
```

Qualche altro 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)
   - Iterando sull'insieme(Esercizio n. 1bis)

b) Uso di `List` 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)
   - scambiare due elementi nella lista (Esercizio 2bis)

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`)

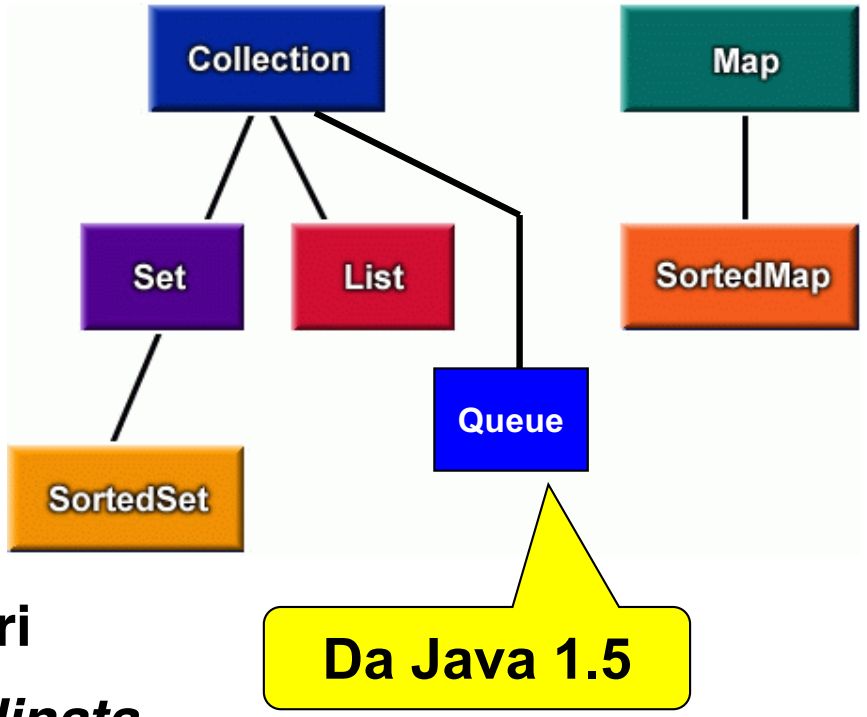
JCF – recap (Lezione in laboratorio)

- **Java Collection Framework (JCF) fornisce il supporto a qualunque tipo di *struttura dati***
  - **Interfacce**
  - **classi che forniscono implementazioni dei vari tipi di strutture dati specificati dalle interfacce**
  - **una classe `Collections` che definisce algoritmi polimorfi (funzioni statiche)**
- **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!)

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*):

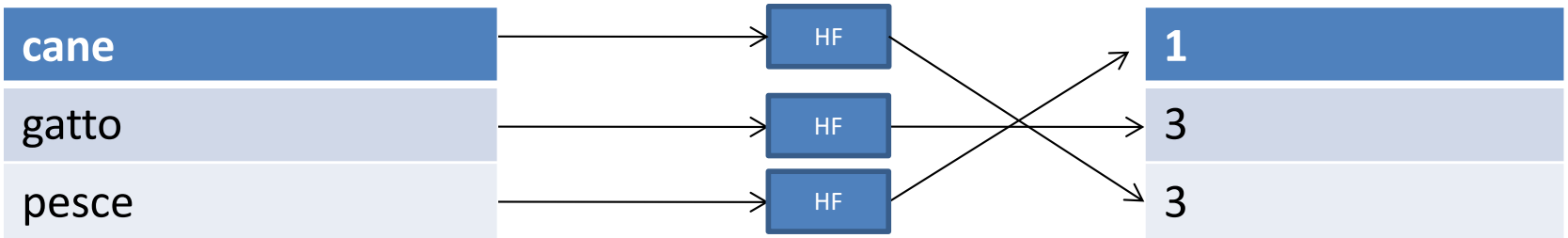
- Liste — O(n)
- Alberi binari di ricerca non bilanciati — O(n)
- Alberi binari di ricerca bilanciati — $O(\log_2 n)$
- **Tabelle hash** — 1

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≤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

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

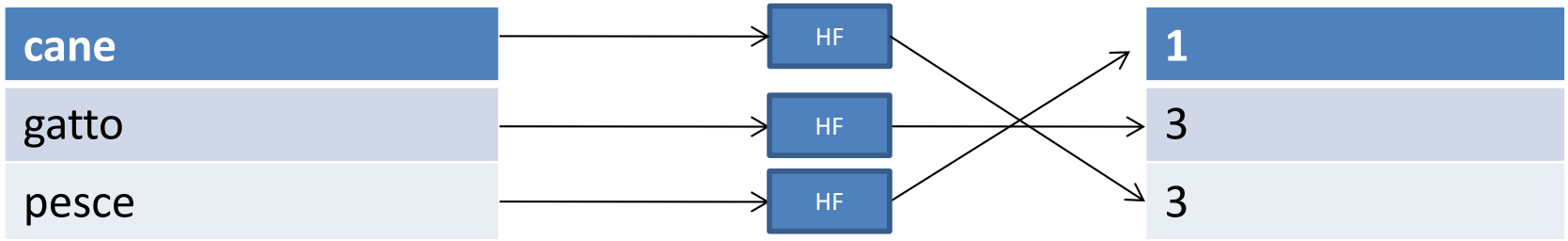
```
import java.util.*;
public class ContaFrequenza {
 public static void main(String args[]) {
   Map m = new HashMap();
   for (int i=0; i<args.length; i++) {
    Integer freq = (Integer) m.get(args[i]);
    m.put(args[i], (freq==null ? new Integer(1) :
                    new Integer(freq.intValue()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

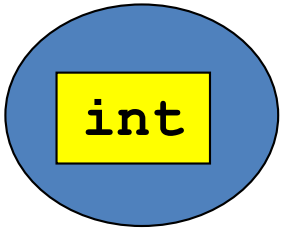
    → problemi di *correttezza* nel type system *(downcasting)*

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*

`boolean`	`Boolean`	`char`	`Character`
`byte`	`Byte`	`short`	`Short`
`int`	`Integer`	`long`	`Long`
`double`	`Double`	`float`	`Float`



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"** (≤ 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"** (≥ 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}
```

JCF: la classe Collections

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*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*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")
    };
    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, 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`

`Vector`, CHI ERA COSTUI?

- **Fino a JDK 1.4, `Vector` era la forma più usata di lista**
  - **all'epoca, la Java Collection Framework non esisteva**
- **Da JDK 1.5, JCF ha reso `List` *la scelta primaria***
  - **metodi con nomi più brevi, con parametri in ordine più naturale**
  - **varietà di implementazioni con diverse caratteristiche**
- **Perciò, anche `Vector` è stato reingegnerizzato per implementare** (a posteriori..) **l'interfaccia `List`**
  - **i vecchi metodi sono stati mantenuti per retro-compatibilità, ma sono stati deprecati**
  - **usare `Vector` oggi implica aderire all'interfaccia `List`**
  - **il `Vector` così ristrutturato è stato mantenuto anche nella JCF "generica" disponibile da JDK 1.5 in poi.**

`Vector` vs. `List`

- Il vecchio `Vector` offriva metodi come:
  - `setElementAt(`*`elemento, posizione`*`)`
  - `elementAt(`*`posizione`*`)`

DIFETTI:
- nomi di metodi lunghi e disomogenei
- argomento *posizione* a volte come 1° , a volte come 2° argomento

- mentre il nuovo `Vector`, aderente a `List`, usa:
  - `set(`*`posizione, elemento`*`)`
  - `get(`*`posizione`*`)`

- Nomi brevi, chiari e coerenti con `Collection`
- argomento posizione sempre come 1° argomento

`Vector` vs. `List` – ESEMPIO

ESEMPIO:
moltiplicazione fra due elementi di un "vettore"

- **in un array scriveremmo semplicemente:**

```
v[i] = v[j].mul(v[k]);
```

- **nel vecchio `Vector` avremmo scritto, cripticamente:**

```
v.setElementAt(
    v.elementAt(j).mul(v.elementAt(k)), i);
```

- **nel nuovo `Vector` si può invece scrivere:**

```
v.set(i, v.get(j).mul(v.get(k)) );
```

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. 2 (`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. 3 e 4 (`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**
- **`Vector`: versione reingegnerizzata e sincronizzata di arrayList**

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. 5 (`Map`) si può scegliere fra:

- **`HashMap`: tabella non ordinato, tempo d'accesso costante**
- **`TreeMap`: tabella ordinato, 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}
```

UNA SCELTA.. DISCUTIBILE: OPERAZIONI "OPZIONALI"

- **Per mantenere ridotto il numero di interfacce, si è scelto di *NON definire una nuova interfaccia per ogni variante* di una data struttura dati.**
- **Si è preferito marcare alcune operazioni come *opzionali***
  - **una implementazione non è tenuta a fornirle: se non le fornisce, deve lanciare una *UnsupportedOperationException.***
  - **tuttavia, tutte le implementazioni fornite dalla JCF supportano *tutte* le operazioni dichiarate dalle interfacce.**
- **Per facilitare l'implementazione di queste interfacce, sono spesso disponibili *classi astratte* contenenti implementazioni *parziali ("skeleton implementations")***

JCF "classica": limiti e problemi

JCF "CLASSICA": PROBLEMI

- **Usare il tipo `Object` per definire *contenitori generici* causa in realtà *vari problemi***
  - **equivale ad abolire il controllo di tipo: ciò rende possibili *operazioni sintatticamente corrette***
    - ossia, che si compilano normalmente
  - **ma *semanticamente errate***
    - ossia, che danno luogo a errori a run-time
- **Conseguenza: *il linguaggio non è "type safe",* ossia *la correttezza è affidata a "commenti sul corretto uso".***

ESEMPIO (1/3)

Supponiamo che una classe definisca la funzione:

```
public static void fillMyDoubleList(List dlist){
    // presuppone che dlist sia una lista di Double
    for(int i=1; i<=16; i++){
        dlist.add(new Double(Math.sqrt(i++)));
    }
}
```

- **Questa funzione, *come dice il commento*, presuppone che le venga passata una lista di `Double`**
- **Tuttavia, il tipo del parametro formale è semplicemente `List` (sottinteso: di `Object`): *nulla esprime formalmente il vincolo che la lista sia composta di soli `Double` !***

ESEMPIO (2/3)

Coerentemente, un cliente dovrebbe usarla così:

```
List listOfDouble = new LinkedList();
fillMyDoubleList(listOfDouble);
for (Object x : listOfDouble) {
  double d = (Double) x; // OK.. in teoria!
}
```

VINCOLO SEMANTICO INESPRESSO: la lista conterrà solo dei Double

ma.. *cosa succede se un cliente invece la usa così ??*

```
List listOfIntegers = new LinkedList();
fillMyDoubleList(listOfIntegers);
for (Object x : listOfIntegers) {
  int k = (Integer) x; // SI COMPILA, MA...
}
```

VINCOLO SEMANTICO VIOLATO, ma nessuno se ne accorge..!!

ESEMPIO (3/3)

A run time, però, succede il disastro!

```
Exception in thread "main"
java.lang.ClassCastException:
java.lang.Double cannot be cast to java.lang.Integer
        at Esempio0.test1(Esempio0.java:30)
        at Esempio0.main(Esempio0.java:14)
```

PERCHÉ È SUCCESSO?

Sebbene formalmente tutto si compili, *in realtà è stato violato un vincolo di progetto,* perché:

- **il progettista di `fillMyDoubleList` ipotizzava che la lista fosse destinata a contenere solo valori `Double`**
- **il progettista del main ha passato però una lista destinata a contenere degli `Integer` *e l'ha usata come tale***

IL NOCCIOLO DEL PROBLEMA

```
List listOfIntegers = new LinkedList();
fillMyDoubleList(listOfIntegers);
for (Object x : listOfIntegers) {
  int k = (Integer) x; // SI COMPILA, MA...
}
```

- Il problema non sta ovviamente nell'aver passato alla funzione `fillMyDoubleList` una lista chiamata `listOfIntegers`, ma nel fatto che *questo nome sottintendeva un'ipotesi di progetto* che così facendo è stata violata.
- Ergo, quando poi il progettista ha *usato quell'ipotesi per accedere alla lista, è accaduto il disastro,* perché l'assunto che essa contenesse solo valori interi – cruciale per la correttezza del cast – si è rivelato falso.

CONCLUSIONE

Nonostante la compilazione si sia svolta correttamente, indicando che non vi erano errori, in realtà *il programma era comunque sbagliato.*

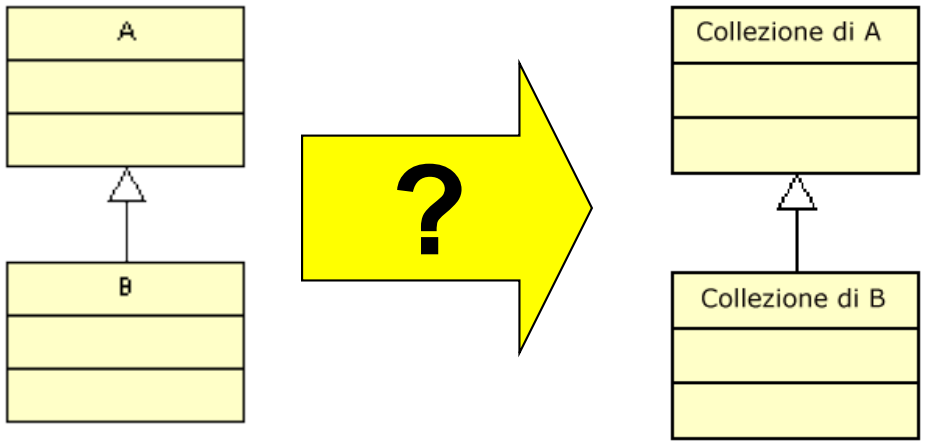
MORALE: Il linguaggio non è "type safe"
(sicuro sotto il profilo dei tipi) ,
perché non garantisce che un programma che passi la compilazione sia corretto!

Purtroppo, non è finita qui:
ulteriori problemi nascono con l'ereditarietà.

UN ALTRO PROBLEMA

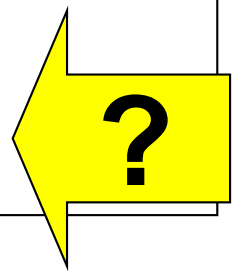
DOMANDA: se il tipo B è un sottotipo di A, è lecito concludere che, di conseguenza, il tipo "collezione di B" *sia un sottotipo* della "collezione di A" ?

Ad esempio: poiché `Integer` deriva `Object`, è lecito dire che, di conseguenza, gli "array di `Integer`" *sono un sottotipo* degli "array di `Object`" ...?



ESPERIMENTO

```
Integer[] arrayOfIntegers = new Integer[4];
arrayOfIntegers[0] = new Integer(12);
Object[] arrayOfObjects = arrayOfIntegers;
```



- Se questo assegnamento passa, significa che per Java gli array di `Integer` ***sono effettivamente un sottotipo*** degli array di `Object`.
- Proviamo:

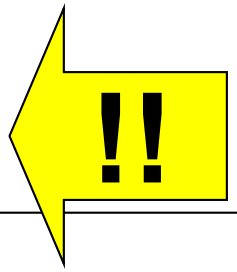
```
Output completed (0 sec consumed) - Normal Termination
```

  - Dunque, per Java gli array di `Integer` ***sono un sottotipo*** degli array di `Object`.
  - ***MA.. sarà la scelta giusta?? Cosa implica?***

IL PROBLEMA

Cosa succede però se adesso si scrive.. *questo?*

```
Integer[] arrayOfIntegers = new Integer[4];
arrayOfIntegers[0] = new Integer(12);
Object[] arrayOfObjects = arrayOfIntegers;
...
arrayOfObjects[1] = "ciao";
```



- **Poiché una stringa è un `Object`, *l'assegnamento è formalmente corretto, ma non ha senso,* perché l'array "vero" che c'è sotto è di `Integer` e non può ospitare stringhe!**
- **CONSEGUENZA: come prima, la compilazione passa, ma poi..**

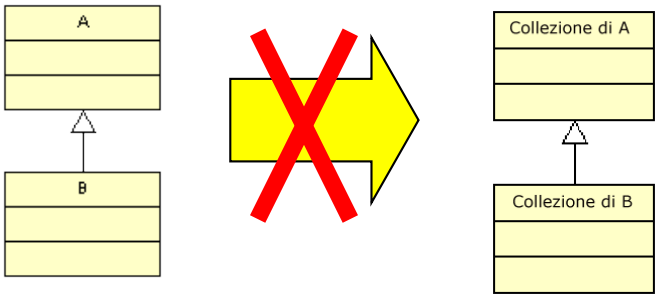
```
Exception in thread "main"
java.lang.ArrayStoreException: String
```

CONCLUSIONE

Di nuovo, nonostante la compilazione si sia conclusa senza errori, *il programma è sbagliato.*

> ***Di nuovo, Il linguaggio non è "type safe" neppure sotto il profilo dell'ereditarietà***

Evidentemente, non è lecito dedurre che, se il tipo B è un sottotipo di A, allora il tipo "collezione di B" è un sottotipo del tipo "collezione di A" .



Serve un approccio *totalmente nuovo.*