



Collection

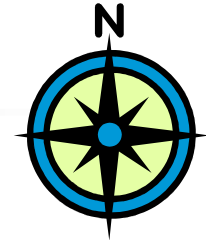




Collection

- L'interfaccia *Collection* modella **insiemi**.
- E' la root della gerarchia delle *Collection*
- Le Collection accettano e gestiscono SOLO oggetti.
- Collection prevede generici metodi per
 - aggiungere, rimuovere, cercare elementi
 - verificare la presenza e il numero di elementi
 - recuperare il “navigatore” della Collection
- Non esiste nessuna implementazione diretta
 - esistono delle sottointerfacce e di queste esistono delle implementazioni
- Alcune Collection consentono i duplicati, altre invece no.

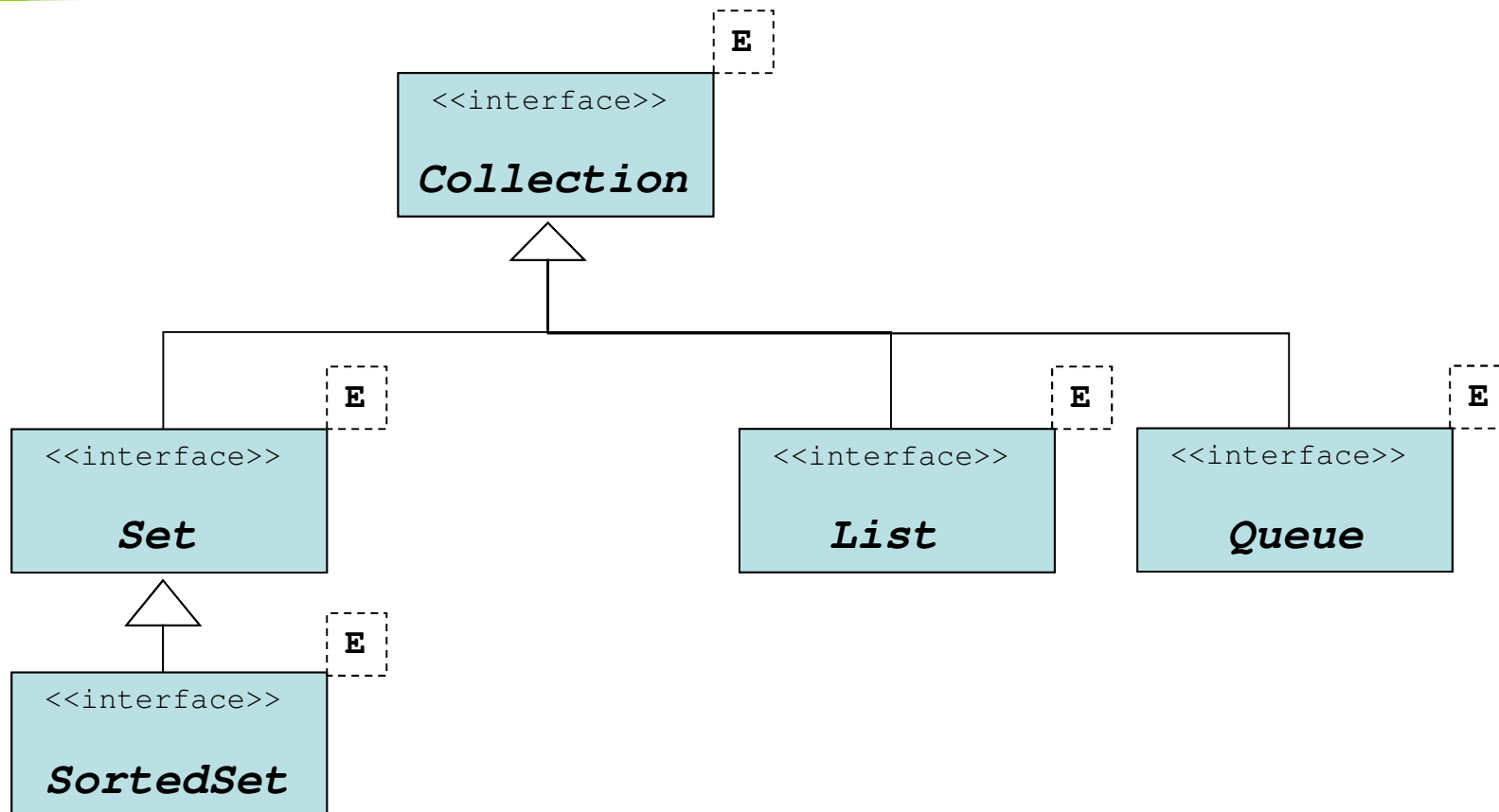
Iterator



- ***Iterator*** è un'interfaccia del pacchetto `java.util`
- Consente la navigazione all'interno di una `Collection`.
- Tutte le `Collection` dispongono di un `Iterator`.
- L' `Iterator` di una `Collection` si ottiene invocando il metodo
`iterator()`
- I metodi dell'interfaccia *Iterator* sono:
 - **`boolean hasNext()`**
 - Verifica se esiste un elemento
 - **`Object next()`**
 - Sposta l'iteratore sul prossimo elemento
 - **`void remove()`**
- NB: quando un `Iterator` ha terminato l'iterazione su una `Collection`, non si può usarlo per una nuova iterazione (bisogna ricrearlo!)



Architettura delle Collection



Per ognuna di queste interfacce ci sono diverse implementazioni concrete



Interface List

```
public interface List extends Collection
```

- Collezione sequenziale di oggetti
- L'accesso agli elementi è tramite indice
- Ammette duplicati
- Alle funzionalità ereditate da **Collection**, vengono aggiunte funzionalità specifiche per l'inserimento e la ricerca.
- C'è un iteratore speciale (specializza **Iterator**) **ListIterator** che permette lo scorrimento bidirezionale



Le Liste

- Le implementazioni note di `List` sono:
 - `ArrayList<E>`
 - `Vector<E>`
 - `LinkedList<E>`
- Il simbolo `E` rappresenta il generico elemento che si può (si dovrebbe) indicare quando si crea una collezione concreta.



Class ArrayList

Implementa List attraverso array di dimensione variabile

❑ Costruttori:

- **ArrayList()** Costruisce un array di dimensione iniziale 10
- **ArrayList(Collection<E> c)** Costruisce un array a partire da una collezione data di oggetti
- **ArrayList(int initialCapacity)** Costruisce un array vuoto specificando la capacità iniziale.



Class ArrayList

☐ Metodi

Per l'inserimento

- `public void add(int index, E element)`
 - può eseguire uno shift e solleva `IndexOutOfBoundsException` (se `index < 0` opp `index > size`)
- `public boolean add(E element)`
- `public boolean addAll(Collection<E> c)`
- `public boolean addAll(int index, Collection<E> c)`
- `public E set(int index, E element)`
 - sostituisce, non esegue shift e solleva
 - `IndexOutOfBoundsException` (se `index > size`)
 - `ArrayIndexOutOfBoundsException` (se `index < 0`)

Per la lettura

- `public E get(int index)`



Class ArrayList

❑ Metodi

Per la ricerca

- `public int indexOf(E element)`
- `public int lastIndexOf(E element)`
- `public boolean contains(E element)`

Per la rimozione

- `public E remove(int index)`
 - Rimuove l'elemento alla posizione indicata ed esegue eventualmente uno shift a sinistra
 - Solleva eccezioni come metodo `set(int, E)`
- `public boolean remove(E element)`
 - Rimuove la prima occorrenza dell'elemento indicato ed esegue eventualmente uno shift a sinistra.
 - Torna true o false a seconda se ha cancellato oppure no.
- `public void clear()`
- `public void trimToSize()`



Class ArrayList

Altri Metodi

- `public int size()`
- `public boolean isEmpty()`
- `public Object[] toArray()`



Il metodo equals

- Per utilizzare metodi come `contains(obj)` e `remove(obj)`, bisogna specificare quando due elementi risultano uguali.
- Se gli elementi della lista sono di un tipo (una classe) definito dal programmatore, ALLORA all'interno di quella classe bisogna ridefinire il metodo

`public boolean equals(Object)`

Esempio:

```
public class Studente{
    private int matricola;

    public boolean equals(Object obj) {
        return obj != null && obj instanceof Studente &&
            ((Studente)obj).matricola == this.matricola;
    }
}
```



Metodo equals

- Quando si esegue l'overriding di `equals()`
 - bisogna invocare `instanceof` per assicurarsi di valutare le classi in modo appropriato
 - bisognerebbe confrontare gli attributi più significativi dell'oggetto
- Proprietà chiave del “contratto” di `equals()`
 - riflessiva → `x.equals(x)` è `true`
 - simmetrica → `x.equals(y)` è `true` → `y.equals(x)` è `true`
 - transitiva → `x.equals(y)` è `true` e `y.equals(z)` è `true` → `x.equals(z)` è `true`
 - consistente → `x.equals(y)` deve restituire sempre lo stesso risultato (anche su multiple chiamate)
 - null → `x != null` → `x.equals(null)` è `false`



Class Vector

- Esiste dalla versione 1.0 di java e funziona essenzialmente come ArrayList (disponibile dalla 1.2)
- Implementa List attraverso array di dimensione variabile.
- A differenza dell'ArrayList, supporta la sincronizzazione.

❑ Costruttori:

- Hanno la stessa firma e funzionamento di quelli di ArrayList
- **Vector(int initialCapacity, int capacityIncrement)**: Costruisce un vettore vuoto specificando la capacità iniziale e l'incremento.



Class LinkedList

- Implementa List attraverso una lista linkata
- Oltre ai metodi classici di List, prevede metodi specifici per
 - inserire, leggere e rimuovere oggetti all'inizio e alla fine della lista
- E' la classe ideale per gestire code e pile
- I costruttori sono simili a quelli di ArrayList
- Come ArrayList, non supporta la sincronizzazione



Class LinkedList

❑ Metodi specifici

- `public void addFirst(E element)`
- `public void addLast(E element)`
- `public E getFirst()`
- `public E getLast()`
- `public E removeFirst()`
- `public E removeLast()`



Collection e Generics

- I generics sono una caratteristica di molti linguaggi di programmazione, sono disponibili in **Java** dalla **1.5**
- Consentono di creare collezioni o contenitori che memorizzano solo il tipo di oggetti specificato
- Il tipo di oggetto si indica tra parentesi angolari "<>"

```
CollectionType<Tipo> collection =  
                                new CollectionType<Tipo>();
```

- Non è possibile inserire oggetti di natura completamente diversa da quelli indicati
- Vantaggi:
 - non sono più necessari cast per estrarre gli oggetti dalla collezione, se sono del tipo definito
 - si evita l'eccezione di **ClassCastException**

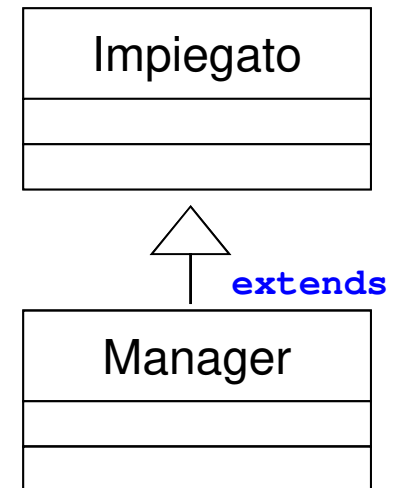
Esempio d'uso

```
// creo un'arraylist di Impiegati
ArrayList<Impiegato> elencoDipendenti =
    new ArrayList<Impiegato>();

// carico un impiegato e un manager:OK
elencoDipendenti.add(new Impiegato(...));
elencoDipendenti.add(new Manager(...));

// Non posso caricare oggetti non presenti nella
// gerarchia di Impiegato: ERRORE di compilazione
elencoDipendenti.add(new Date(...));

// recupero dell'impiegato SENZA cast
Impiegato imp = elencoDipendenti.get(0);
// recupero del manager CON cast
Manager mng = (Manager)elencoDipendenti.get(1);
```



NOTA: l'esempio usa ArrayList, ma i generics si usano su tutte le Collection (sia per la gerarchia List che per quella Set)



Il costrutto foreach

Dalla JSE 5, le collection che usano i generics possono usare una versione “evoluta” del costrutto **for**, il costrutto **foreach**

```
for( Element element: collection) {... ..}
```

1) Si dichiara

- il generico elemento → **element**
- l'insieme su cui iterare → **collection**

2) **Non** bisogna:

- dichiarare e incrementare indici
- ottenere e usare un Iterator

NB: Si può usare anche con gli array



Esempio d'uso

```
LinkedList<Impiegato> lista = new LinkedList<Impiegato>();

lista.add(new Impiegato("mario",1500, new Date()));
lista.add(new Impiegato("gino",1200, new Date()));
lista.add(new Impiegato("luca",1100, new Date()));

for(Impiegato dipendente : lista)
{
    System.out.println("Dipendente: " + dipendente );
}
```



AutoBoxing & UnBoxing

- Problematiche:
 - “Conversioni da primitivi a wrapper relativi”
 - “Le Collection di Java supportano solo oggetti e non primitivi”

- La soluzione:

L'**autoboxing** consente:

- assegnare primitivi a wrapper
- il caricamento automatico di primitivi in una collection

L'**unboxing** consente:

- assegnare wrapper a primitivi
- il recupero del wrapper (creato dal primitivo) e caricato sulla collection

- Cosa avviene in automatico?
 - La costruzione del tipo wrapper (prima del caricamento)
 - Il cast (per la lettura)



Esempio d'uso

```
// autoboxing
Integer wrapper = 5;
```

```
// unboxing
int i = new Integer(5);
```

```
// autoboxing sulla Collection
List <Double> numeri = new LinkedList <Double> ();
numeri.add(1.5);
numeri.add(3.61);
numeri.add(12.722);
```

La collection **numeri** carica
e recupera **Double**

```
// unboxing sulla collection
for(int i=0; i<numeri.size(); i++)
{
    System.out.println("element " + i + ": " + numeri.get(i));
}
```



Esercizi sulle liste

- Data la classe `Impiegato`, creare una classe `Azienda` che ne faccia la gestione. (usare `ArrayList`)
 - Metodi proposti:
 - `assumi`
 - `licenzia`
 - `incrementaSalarioPerTutti`
- Creare una classe `Pila` che modelli una struttura LIFO (usare `LinkedList`)
 - Metodi proposti:
 - `add(Object ob)`
 - `remove(Object ob)`

con eventuale gestione delle eccezioni per la pila vuota

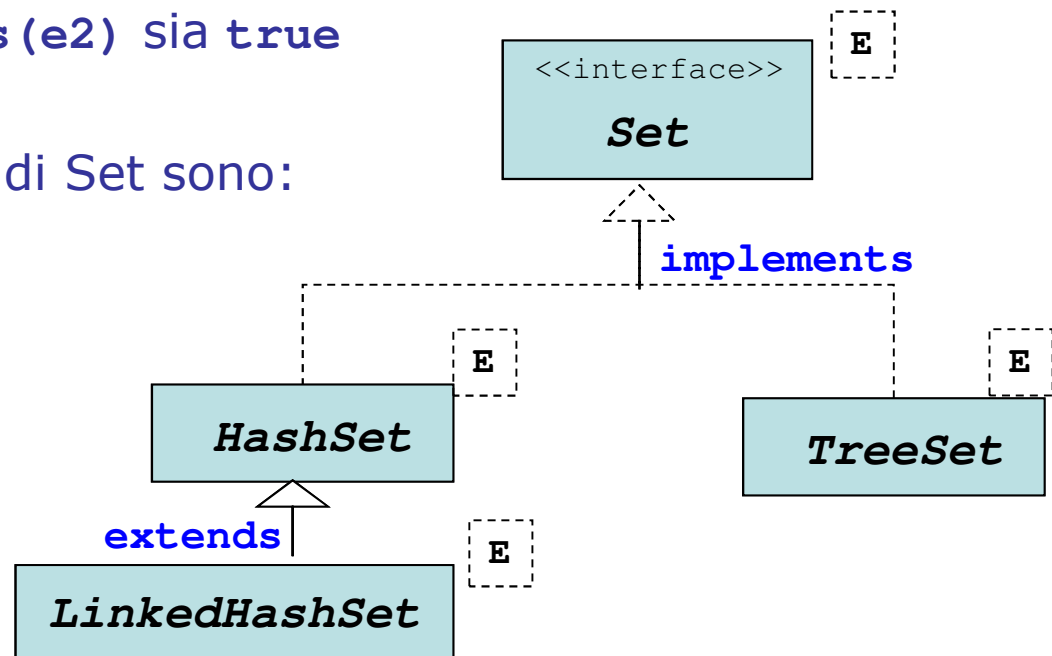
Interface Set

`public interface Set extends Collection`

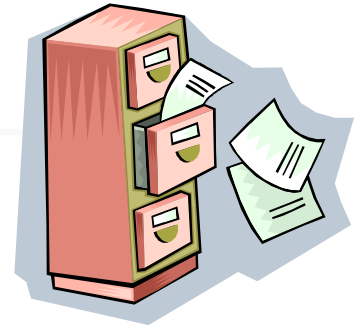
- Modella gli insiemi matematici
- Rappresenta una Collection che non accetta elementi ripetuti.
- Formalmente un Set non contiene una coppia di elementi e_1, e_2 tali che `e1.equals(e2)` sia `true`

- Le implementazioni note di Set sono:

- HashSet
- LinkedHashSet
- TreeSet



Class HashSet



- Si occupa della gestione di insiemi senza ripetizioni **ma non ordinati**
- La struttura dati che lo rappresenta è un array i cui elementi sono liste, ciascuna lista è un *bucket*
- Gli oggetti dell'insieme dovrebbero ridefinire:
 - il metodo **boolean equals (Object)** – per distinguere “doppioni”
 - il metodo **int hashCode ()** – per “gestione dei buckets”
tale metodo deve rispettare il “contract” secondo cui

Se `x.equals(y)` è **true** allora `x.hashCode () == y.hashCode ()`

NB: prima viene chiamato **hashCode ()**, poi se ci sono oggetti con lo stesso hashcode, si invoca anche **equals ()**

- Il numero di bucket totali e il fattore di carico per ciascun bucket si può impostare col costruttore (valori default 16 e 0.75)



Metodo hashCode

- Proprietà chiave del “contratto” di **hashCode()** :
 - consistente → **x.hashCode()** deve restituire sempre lo stesso numero intero (anche su multiple chiamate)
 - **x.equals(y) true** ➡ **x.hashCode() == y.hashCode()**
 - **x.equals(y) false**, **NON** è detto che
x.hashCode() != y.hashCode()



equals e hashCode

- Se `x.equals(y)` è `true` ➡ `x.hashCode() == y.hashCode()`
- Quindi oggetti uguali per `equals()` hanno stesso `hashCode`
- E dunque è anche vero che
 - oggetti che NON hanno stesso `hashCode`, NON devono essere uguali per `equals()`
- Se si sovrascrive **`equals`** bisogna sovrascrivere anche **`hashCode`**
- Le classi che usano chiavi hash sono:
 - `HashMap`, `HashSet`, `Hashtable`
 - `LinkedHashMap` e `LinkedHashSet`
- Un metodo `hashCode` efficiente, distribuisce uniformemente le chiavi hash attraverso i buckets
- Il metodo `hashCode` può tornare uno stesso valore per tutti gli oggetti, ma è davvero poco efficiente!



Esempio d'uso di HashSet

Voglio creare un `HashSet` di `String`.

Le stringhe hanno un valido overriding di `equals` e di `hashCode`.

```
HashSet<String> ha = new HashSet<String>();  
ha.add("serpente");  
ha.add("ape");  
ha.add("farfalla");  
ha.add("farfalla");           // NON viene aggiunto!  
ha.add("furetto");  
ha.add("gattino");
```

```
for (String entry: ha) {  
    System.out.println("elemento: " + entry);  
}
```

L'iteratore mostrerà:

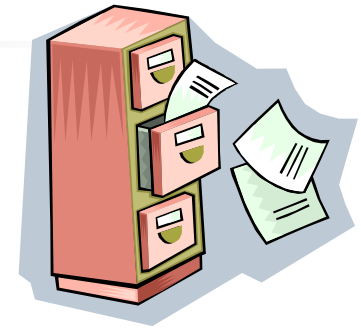
serpente
gattino
farfalla
ape
furetto

Gli hashcode sono:

serpente → 1373560042
gattino → -188685104
farfalla → 927458191
ape → 96790
furetto → -505888915

L'ordine di navigazione NON è prevedibile!

Class LinkedHashSet



- E' un sottotipo di HashSet che consente la **navigazione** degli elementi secondo **l'ordine d'inserimento**
- La struttura dati che lo rappresenta è ancora un array di *bucket*
- Gli oggetti dell'insieme dovrebbero ridefinire:
 - il metodo **boolean equals(Object)** – per distinguere “doppioni”
 - il metodo **int hashCode()** – per “gestione dei buckets”
tale metodo deve rispettare il “contract” secondo cui

Se `x.equals(y)` è **true** allora `x.hashCode() == y.hashCode()`

NB: prima viene chiamato **hashCode()**, poi se ci sono oggetti con lo stesso hashcode, si invoca anche **equals()**

- Il numero di bucket totali e il fattore di carico per ciascun bucket si può impostare col costruttore (valori default 16 e 0.75)



Esempio d'uso di LinkedHashSet

```
LinkedHashSet<String> lha = new LinkedHashSet<String>();  
lha.add("serpente");  
lha.add("ape");  
lha.add("farfalla");  
lha.add("farfalla");    // NON viene aggiunto!  
lha.add("furetto");  
lha.add("gattino");  
  
for (String entry: lha) {  
    System.out.println("elemento: " + entry);  
}
```

NB: Gli elementi nei bucket sono disposti
SEMPRE secondo il valore dell'hashCode

Ma l'ordine di navigazione è prevedibile!

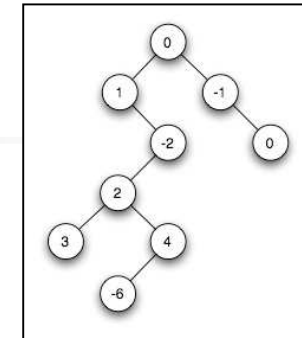
L'iteratore mostrerà:

serpente
ape
farfalla
furetto
gattino



Ultimo inserito

Class TreeSet



- Si occupa della gestione di insiemi **ordinati** mediante alberi binari
- Gli oggetti dell'insieme **devono** prevedere un criterio di confronto che verrà implementato
 - nel metodo **compareTo** previsto dall'interfaccia **Comparable**
 - nel metodo **compare** previsto dall'interfaccia **Comparator**
- Un TreeSet istanziato con **TreeSet ()** gestirà oggetti **Comparable** → userà **compareTo**
- Un TreeSet istanziato con **TreeSet (Comparator)** gestirà oggetti attraverso un **Comparator** → userà **compare**
- Se gli elementi dell'insieme sono stringhe o tipi wrapper, per essi è già definito un ordine naturale



Esempio d'uso di TreeSet

Voglio creare un `TreeSet` di `String`.

Le stringhe sono confrontabili e il criterio definito è quello che induce l'ordinamento alfabetico

```
TreeSet<String> set = new TreeSet<String>();  
set.add("dog");  
set.add("ant");  
set.add("horse");  
set.add("gorilla");
```

```
for (String element: set) {  
    System.out.println(element);  
}
```

L'iteratore mostrerà:

```
ant  
dog  
gorilla  
horse
```

Indipendentemente da come vengono inseriti nell'insieme, vengono iterati sempre mantenendo l'ordine stabilito



TreeSet is_a SortedSet

TreeSet implementa **SortedSet** quindi oltre ai metodi di **Set**, dispone anche di:

- **SortedSet<E> subSet(E fromElement, E toElement)**

torna una vista del sottoinsieme ottenuto partendo dal 1° Object fino al 2°, con quest'ultimo NON compreso ($\geq 1^\circ$ e $< 2^\circ$)

– *Se from > to* → [java.lang.IllegalArgumentException](#)

- **SortedSet<E> headSet(E toElement)**

torna una vista del sottoinsieme ottenuto partendo **dalla “testa”** fino all'elemento specificato escluso ($<$ parametro)

- **SortedSet<E> tailSet(E fromElement)**

torna una vista del sottoinsieme ottenuto partendo dall'elemento specificato **fino alla “coda”** (\geq parametro)

NB: i sottoinsiemi ottenuti rimangono collegati con quello iniziale → modifiche al 1° comportano modifiche anche al 2° (coerentemente con le regole della sua creazione).



Backed Collection

Le collezioni che rimangono “fuse” insieme si chiamano **Backed Collection**

Esempio:

```
TreeSet<String> set = new TreeSet<String>();
set.add("ant"); set.add("dog"); set.add("gorilla"); set.add("horse");
SortedSet<String> subset;
subset = set.subSet("bat", "gorilla"); // bat incluso, gorilla escluso

System.out.println(set + " " + subset); // stampa le 2 collezioni
```

Stamperà:

```
{ant, dog, gorilla, horse}      {dog}
```

```
set.add("bat"); // aggiunge un elemento a set
subset.add("fish"); // aggiunge un elemento a subset
set.add("zebra"); // aggiunge a set, fuori range → OK
```

```
System.out.println(set + " " + subset); // infine stampa le 2 collection
```

Stamperà:

```
{ant, bat, dog, fish, gorilla, horse, zebra}      {bat, dog, fish}
```

```
subset.add("pig"); // aggiunge a subset, fuori range → ERRORE
                   java.lang.IllegalArgumentException: key out of
                   range
```



TreeSet is_a NavigableSet

- **TreeSet** possiede alcuni metodi aggiuntivi poiché è un'implementazione di **NavigableSet**:
- **public E first()**
- **public E last()**
- **public E ceiling(E element)**
 - ritorna l'oggetto più piccolo dell'insieme \geq `element`
- **public E floor(E element)**
 - ritorna l'oggetto più grande dell'insieme \leq `element`
- **public E higher(E element)**
 - ritorna l'oggetto più piccolo dell'insieme $>$ `element`
- **public E lower(E element)**
 - ritorna l'oggetto più grande dell'insieme $<$ `element`

Esempio d'uso

Creo un TreeSet di Integer e lo popolo.

```
TreeSet<Integer> treeset = new TreeSet<Integer>();  
treeset.add(new Integer(2));  
treeset.add(new Integer(5));  
treeset.add(new Integer(7));  
treeset.add(new Integer(33));
```



```
System.out.println("Floor: "+treeset.floor(6));  
System.out.println("Ceiling: "+treeset.ceiling(6));  
System.out.println("Higher: "+treeset.higher(6));  
System.out.println("Lower: "+treeset.lower(6));
```

Mostrerà

Floor: 5

è il numero + grande minore/uguale di 6

Ceiling: 7

è il numero + piccolo maggiore/uguale di 6

Higher: 7

è il numero + piccolo maggiore di 6

Lower: 5

è il numero + grande minore di 6

Invece floor(7) è 7, ceiling(7) è 7, higher(7) è 33, lower(7) è 5






Queue e PriorityQueue

public interface **Queue** extends **Collection**

- Rappresenta una struttura di elementi di tipo FIFO.
- Una sua implementazione è **PriorityQueue**
- E' una coda con accesso **ordered rispetto alla priorità** degli oggetti (la priorità è data dal metodo **compareTo**)
- La struttura è quella di una classica coda, ma l'accesso non rispetta la logica FIFO bensì applica il criterio della priorità.
- Precisamente:
 - Se gli oggetti vengono recuperati con Iterator (o tramite foreach), risulteranno posizionati in un ordine non prevedibile
 - Se gli oggetti vengono rimossi dalla coda, risulteranno in ordine di priorità



Metodi di PriorityQueue

- Oltre ai metodi Collection, le code dispongono anche di:
-  **E peek()** → *equivalente a LinkedList.getFirst*
legge l'elemento in **testa**
-  **void offer(E)** → *equivalente a LinkedList.addLast*
aggiunge in **coda** l'elemento specificato.
NB: funziona come add(E)
-  **E poll()** → *equivalente a LinkedList.removeFirst*
rimuove l'elemento dalla **testa**



Esempio di PriorityQueue

Creo una `PriorityQueue` di oggetti `Museo`, per cui è definito un criterio di ordinamento/priorità basato sull'ordine alfabetico della location (del museo):

`offer` carica e `poll` rimuove dalla testa

```
PriorityQueue<Museo> qq = new PriorityQueue<Museo> ();  
qq.offer(new Museo ("Louvre", "Paris"));  
qq.offer(new Museo ("Uffizi", "Firenze"));  
qq.offer(new Museo ("Capodimonte", "Napoli"));  
qq.offer(new Museo ("El Prado", "Madrid"));
```

```
System.out.println(qq.poll().getLocation());  
System.out.println(qq.poll().getLocation());  
System.out.println(qq.poll().getLocation());  
System.out.println(qq.poll().getLocation());
```

Stamperà:

Firenze
Madrid
Napoli
Paris



Esempio di PriorityQueue(2)

Sulla stessa `PriorityQueue` di oggetti `Museo`, eseguo invece una navigazione con `Iterator` (utilizzo il `foreach`)

```
for(Museo museo: qq)
    System.out.println(museo.getLocation());
```

Si ottiene una sequenza che
non segue nessun criterio in particolare

- potrebbe capitare che li cicla con ordine
che sembra uguale al criterio, ma è un caso!!



Stamperà:

Firenze
Madrid
Paris
Napoli



Esercizio: Il lotto



- Scrivere una classe col main che gestisca l'estrazione del lotto utilizzando un `HashSet`. Per semplicità si assume di avere una sola ruota.
- Bisogna gestire:
 - L'estrazione casuale di 5 numeri interi diversi compresi tra 1 e 90
 - La giocata del giocatore → minimo 1 numero, massimo 5 numeri diversi tra 1 e 90
 - La verifica della vittoria

Esercizio: il Museo



- Scrivere un programma che consenta la gestione delle opere di un museo.
- Le opere possono essere:
 - Quadri
 - Statue

I quadri hanno codice (univoco), titolo, autore e tecnica usata.
Le statue invece codice (univoco), titolo, autore, materiale e altezza.
- Le opere che vengono portate al museo possono essere esposte oppure messe in deposito (se non c'è posto)
- Il deposito ha spazio "infinito", il museo invece ha un numero massimo di opere che può esporre.
- Un'opera è sempre in uno di questi 2 stati:
 - Esposta
 - Non esposta, cioè è in deposito

Museo: descrizione



- Le funzionalità da implementare sono:
 - carica(Opera opera): void
questo metodo deve verificare se c'è posto nel museo, nel caso mette lo stato dell'opera ad **esposta**, viceversa pone l'opera in deposito (e dunque risulterà **non esposta**).
 - sposta(int codice): Opera
sposta l'opera dalle sale del museo nel deposito
 - cerca(int codice) : Opera
cerca l'opera per codice (sia museo che nel deposito)
 - stampaSala(): String
 - stampaDeposito(): String
- Progettare le classi usando le Collection della Sun.
secondo le esigenze