

# 10

## Java Collections Framework

Mirko Viroli  
`mirko.viroli@unibo.it`

C.D.L. Ingegneria e Scienze Informatiche  
ALMA MATER STUDIORUM—Università di Bologna, Cesena

a.a. 2018/2019



## Goal della lezione

- Illustrare la struttura del Java Collections Framework
- Mostrare gli utilizzi delle funzionalità base
- Discutere alcune tecniche di programmazione correlate

## Argomenti

- Presentazione Java Collections Framework
- Iteratori e foreach
- Collezioni, Liste e Set
- HashSet e TreeSet

# Java Collections Framework

## Java Collections Framework (JCF)

- È una libreria del linguaggio Java
- È una parte del package `java.util`
- Gestisce strutture dati (o collezioni) e relativi algoritmi

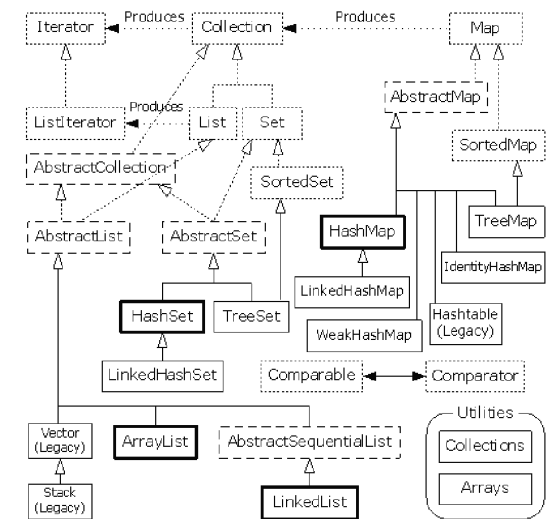
## Importanza pratica

- Virtualmente ogni sistema fa uso di collezioni di oggetti
- Conoscerne struttura e dettagli vi farà programmatori migliori

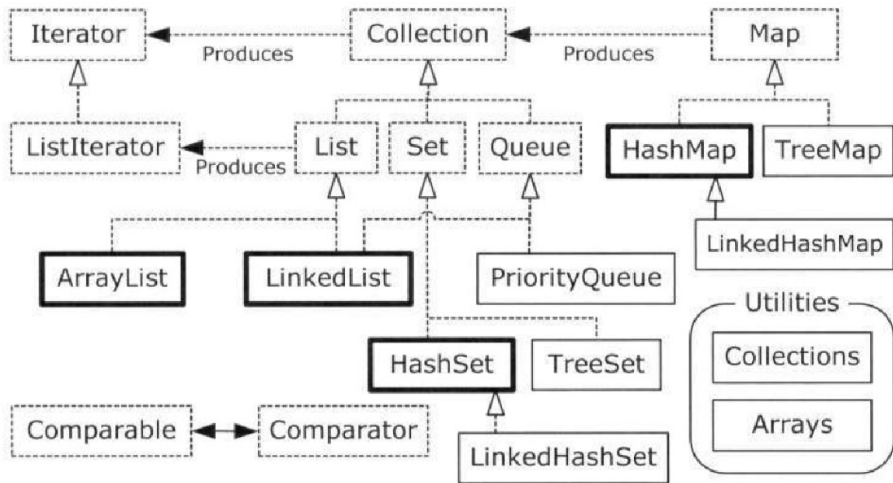
## Importanza didattica

- Fornisce ottimi esempi di uso di composizione, ereditarietà, genericità
- Mette in pratica pattern di programmazione di interesse
- Impatta su alcuni aspetti del linguaggio da approfondire

# JCF – struttura complessiva



# JCF – struttura riorganizzata



# JCF – alcuni aspetti generali

È complessivamente piuttosto articolato

- Un nostro obiettivo è quello di isolare una sua sottoparte di interesse
- Identificando e motivando le funzionalità prodotte

Due tipi di collection, ognuna con varie incarnazioni

- Collection – contenitore di elementi atomici
  - ▶ 3 sottotipi: List (sequenze), Set (no duplicazioni), Queue
- Map – contenitore di coppie chiave-valore

Interfacce/classi di interesse:

- Interfacce: Collection, List, Set, Iterator, Comparable
- Classi collection: ArrayList, LinkedList, HashSet, HashMap
- Classi con funzionalità: Collections, Arrays

# Una nota su eccezioni e JCF

## Eccezioni: un argomento che tratteremo in dettaglio

Un meccanismo usato per gestire eventi ritenuti fuori dalla normale esecuzione (errori), ossia per dichiararli, lanciarli, intercettarli

## JCF e eccezioni

- Ogni collection ha sue regole di funzionamento, e non ammette certe operazioni che richiedono controlli a tempo di esecuzione (ad esempio, certe collezioni sono immutabili, e non si può tentare di scriverci)
- Molti metodi dichiarano che possono lanciare eccezioni – ma possiamo non preoccuparcene per ora



- 1 Iteratori e foreach
- 2 Collection, List, Set
- 3 Implementazioni di Set





# Foreach

## Costrutto foreach

- Abbiamo visto che può essere usato per iterare su un array in modo più astratto (compatto, leggibile)
  - ▶ `for(int i: array){...}`
- Java fornisce anche un meccanismo per usare il foreach su qualunque collection, in particolare, su qualunque oggetto che implementa l'interfaccia `java.lang.Iterable<X>`

## Iterable e Iterator

- L'interfaccia `Iterable` ha un metodo per generare e restituire un (nuovo) `Iterator`
- Un iteratore è un oggetto con metodi `next()`, `hasNext()` (e `remove()`)
- Dato l'oggetto `coll` che implementa `Iterable<T>` allora il foreach diventa:
  - ▶ `for(T element: coll){...}`

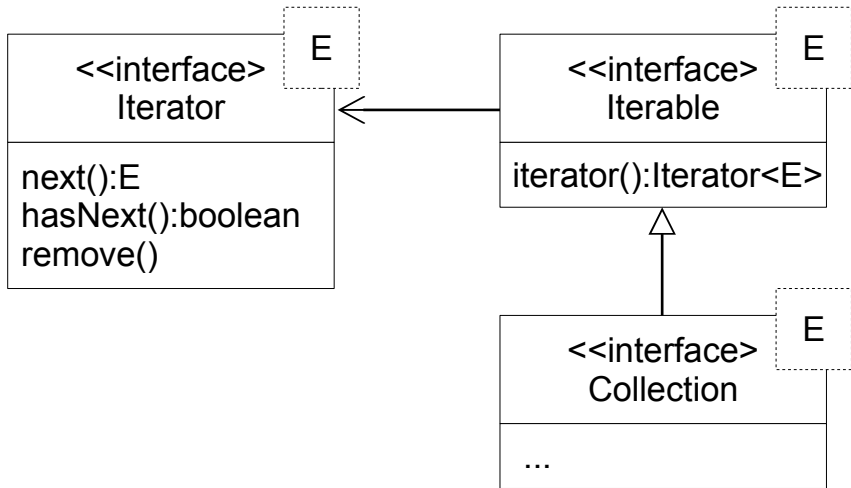
# Interfacce per l'iterazione

```
1 package java.lang;
2 import java.util.Iterator;
3 public interface Iterable<T> {
4     /**
5      * Returns an iterator over a set of elements of type T.
6      *
7      * @return an Iterator.
8      */
9     Iterator<T> iterator();
10 }
```

```
1 package java.util;
2
3 public interface Iterator<E> {
4     boolean hasNext();
5     E next();
6     void remove(); // throws UnsupportedOperationException
7 }
```

```
1 package java.util;
2
3 public interface Collection<E> implements Iterable<E> { .. }
```

# Interfacce per l'iterazione – UML



# Esempio di iterable ad-hoc, e suo uso

```
1 public class Range implements Iterable<Integer>{
2
3     private final int start;
4     private final int stop;
5
6     public Range(final int start, final int stop){
7         this.start = start;
8         this.stop = stop;
9     }
10
11     public java.util.Iterator<Integer> iterator(){
12         return new RangeIterator(this.start, this.stop);
13     }
14 }
```

```
1 import java.util.Iterator;
2
3 public class UseRange{
4     public static void main(String[] s){
5         for (final int i: new Range(5,12)){
6             System.out.println(i);
7             // 5 6 7 8 9 10 11 12
8         }
9     }
```

# Realizzazione del corrispondente iteratore

```
1 class RangeIterator implements java.util.Iterator<Integer>{
2
3     private int current;
4     private final int stop;
5
6     public RangeIterator(final int start, final int stop){
7         this.current = start;
8         this.stop = stop;
9     }
10
11     public Integer next(){
12         return this.current++;
13     }
14
15     public boolean hasNext(){
16         return this.current <= this.stop;
17     }
18
19     public void remove(){}
20 }
```



# Iteratori e collezioni: preview

```
1 import java.util.*;
2
3 public class UseCollectionIterator{
4
5     public static void main(String[] s){
6         // Uso la LinkedList
7         final LinkedList<Double> list = new LinkedList<>();
8         // Inserisco 50 elementi
9         for (int i=0;i<50;i++){
10             list.add(Math.random());
11         }
12         // Stampo con un foreach
13         int ct=0;
14         for (double d: list){
15             System.out.println(ct++ + "\t" + d);
16         }
17         // 0          0.10230513602737423
18         // 1          0.4318582138894327
19         // 2          0.5239222319032795
20         // ..
21     }
22 }
```

- 1 Iteratori e foreach
- 2 Collection, List, Set
- 3 Implementazioni di Set

# Interfaccia Collection

## Ruolo di questo tipo di dato

- È la radice della gerarchia delle collezioni
- Rappresenta gruppi di oggetti (duplicati/non, ordinati/non)
- Implementata via sottointerfacce (`List` e `Set`)

## Assunzioni

- Definisce operazioni base valide per tutte le collezioni
- Assume implicitamente che ogni collezione abbia due costruttori
  - ▶ Senza argomenti, che genera una collezione vuota
  - ▶ Che accetta un `Collection`, dal quale prende tutti gli elementi
- Le operazioni di modifica sono tutte “opzionali”
  - ▶ potrebbero lanciare un `UnsupportedOperationException`
- Tutte le operazioni di ricerca lavorano sulla base del metodo `Object.equals()` da chiamare sugli elementi
  - ▶ questo metodo accetta un `Object`, influenzando su alcuni metodi di `Collection`



# Collection

```
1 public interface Collection<E> extends Iterable<E> {
2
3     // Query Operations
4     int size();                // number of elements
5     boolean isEmpty();         // is the size zero?
6     boolean contains(Object o); // does it contain an element equal to o?
7     Iterator<E> iterator();    // yields an iterator
8     Object[] toArray();        // convert to array of objects
9     <T> T[] toArray(T[] a);    // puts in 'a', or create new if too small
10
11     // Modification Operations
12     boolean add(E e);          // adds e
13     boolean remove(Object o);  // remove one element that is equal to o
14
15     // Bulk Operations
16     boolean containsAll(Collection<?> c); // contain all elements in c
17     ?
18     boolean addAll(Collection<? extends E> c); // add all elements in c
19     boolean removeAll(Collection<?> c); // remove all elements in c
20     boolean retainAll(Collection<?> c); // keep only elements in c
21     void clear();              // remove all element
22
23     // ...and other methods introduced in Java 8
24 }
```

# Usare le collezioni

```
1 public class UseCollection{
2     public static void main(String[] s){
3         // Uso una incarnazione, ma poi lavoro sull'interfaccia
4         final Collection<Integer> coll = new ArrayList<>();
5         coll.addAll(Arrays.asList(1,3,5,7,9,11)); // var-args
6         System.out.println(coll); // [1, 3, 5, 7, 9, 11]
7
8         coll.add(13);
9         coll.add(15);
10        coll.add(15);
11        coll.remove(7);
12        System.out.println(coll); // [1, 3, 5, 9, 11, 13, 15, 15]
13
14        coll.removeAll(Arrays.asList(11,13,15));
15        coll.retainAll(Arrays.asList(1,2,3,4,5));
16        System.out.println(coll); // [1, 3, 5]
17        System.out.println(coll.contains(3)); // true
18        System.out.println(Arrays.toString(coll.toArray()));
19
20        Integer[] a = new Integer[2];
21        a = coll.toArray(a);
22        System.out.println(Arrays.deepToString(a));
23    }
24 }
```

# Set e List

## Set

- Rappresenta collezioni senza duplicati
  - ▶ nessuna coppia di elementi porta `Object.equals()` a dare `true`
  - ▶ non vi sono due elementi `null`
- Non aggiunge metodi rispetto a `Collection`
- I metodi di modifica devono rispettare la non duplicazione

## List

- Rappresenta sequenze di elementi
- Ha metodi per accedere ad un elemento per posizione (0-based)
- Andrebbe scandito via iteratore/foreach, non con indici incrementali
- Fornisce un list-iterator che consente varie operazioni aggiuntive

La scelta fra queste due tipologie non dipende da motivi di performance, ma da quale modello di collezione serva!

# Set e List

```
1 public interface List<E> extends Collection<E> {
2     // Additional Bulk Operations
3     boolean addAll(int index, Collection<? extends E> c);
4
5     // Positional Access Operations
6     E get(int index);                // get at position index
7     E set(int index, E element);     // set into position index
8     void add(int index, E element);  // add, shifting others
9     E remove(int index);             // remove at position index
10
11    // Search Operations
12    int indexOf(Object o);            // first equals to o
13    int lastIndexOf(Object o);       // last equals to o
14
15    // List Iterators
16    ListIterator<E> listIterator();   // iterator from 0
17    ListIterator<E> listIterator(int index); // ..from index
18
19    // View
20    List<E> subList(int fromIndex, int toIndex);
21 }
```

```
1 public interface Set<E> extends Collection<E>{}
```

# ListIterator

```
1 package java.util;
2
3 public interface ListIterator<E> extends Iterator<E> {
4     // Query Operations
5
6     boolean hasNext();
7     E next();
8     boolean hasPrevious();
9     E previous();
10    int nextIndex();
11    int previousIndex();
12
13    // Modification Operations
14
15    void remove();
16    void set(E e);
17    void add(E e);
18 }
```

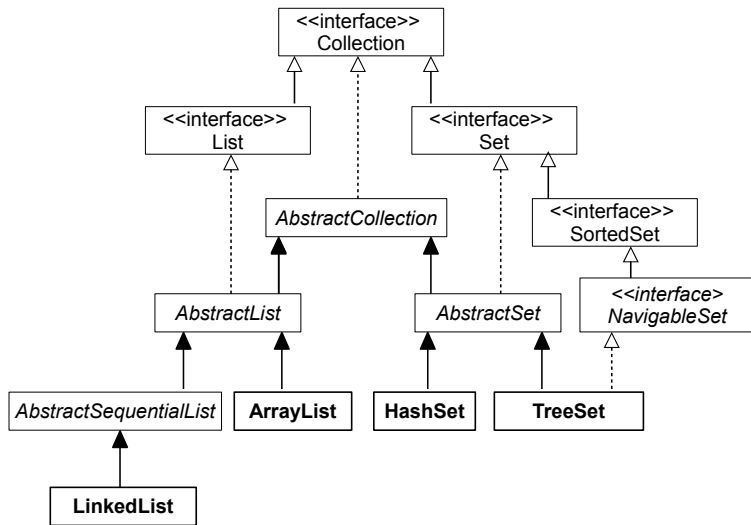


# UseListIterator

```
1 public class UseListIterator{
2     public static void main(String[] s){
3         // Uso una incarnazione, ma poi lavoro sul List
4         final List<Integer> list = new ArrayList<>();
5         list.addAll(Arrays.asList(1,3,5,7,9,11)); // var-args
6
7         final ListIterator<Integer> it = list.listIterator();
8         while (it.hasNext()){
9             it.add(it.next()+1);
10        }
11        System.out.println(list); // [1, 2, 3, ..., 10, 11,12]
12        while (it.hasPrevious()){
13            System.out.println("back: "+it.previous()); // 12 .. 1
14        }
15        for (final int i: list.subList(3,10)){
16            System.out.println("forth - 3 to 10: "+i); // 4 .. 10
17        }
18    }
19 }
```



# Implementazione collezioni – UML



# Implementazione collezioni: linee guida generali

## Una modalità di progettazione da ricordare

- Interfacce: riportano le funzionalità definitorie del concetto
- Classi astratte: fattorizzano codice comune alle varie implementazioni
- Classi concrete: realizzano le varie implementazioni

## Nel codice cliente..

- In variabili, argomenti, tipi di ritorno, si usano le interfacce
- Le classi concrete solo nella `new`, a parte casi molto particolari
- Le classi astratte non si menzionano praticamente mai, solo eventualmente per chi volesse costruire una nuova implementazione





# Implementazione collezioni – Design space

## Classi astratte

- `AbstractCollection`, `AbstractList`, e `AbstractSet`
- Realizzano “scheletri” di classi per collezioni, corrispondenti alla relative interfacce
- Facilitano lo sviluppo di nuove classi aderenti alle interfacce

## Un esempio: `AbstractSet`

- Per set immutabili, richiede solo di definire `size()` e `iterator()`
- Per set mutabili, richiede anche di ridefinire `add()`
- Per motivi di performance si potrebbero fare ulteriori override

## Classi concrete.. fra le varie illustreremo:

- `HashSet`, `TreeSet`, `ArrayList`, `LinkedList`
- La scelta riguarda quasi esclusivamente esigenze di performance

# Esempio di creazione di un nuovo set: RangeSet

```
1 import java.util.*;
2
3 /* Permette di definire un set di valori integer incrementali,
4    senza doverli esplicitamente inserire in memoria, e quindi
5    prediligendo l'occupazione in memoria al tempo d'accesso */
6
7 public class RangeSet extends AbstractSet<Integer> {
8
9     private final int start;
10    private final int stop;
11
12    public RangeSet(final int start, final int stop) {
13        this.start = start;
14        this.stop = stop;
15    }
16
17    public int size() {
18        return (this.stop >= this.start) ? this.start - this.stop + 1 : 0;
19    }
20
21    public Iterator<Integer> iterator() {
22        // Il RangeIterator già visto...
23        return new RangeIterator(this.start, this.stop);
24    }
25 }
```

# Uso di RangeSet

```
1 import java.util.*;
2
3 public class UseRangeSet {
4
5     public static void main(String[] s) {
6         // r è un Set a tutti gli effetti
7         final RangeSet r = new RangeSet(0, 100);
8
9         // ad esempio, lo uso per iterare
10        for (final int i : r) {
11            System.out.println("Elem: " + i);
12        }
13
14        // ad esempio, uso la contains()
15        System.out.println(r.contains(25));
16
17        // è comunque un set immutabile
18        // quindi niente add(), remove(),...
19    }
20 }
```



- 1 Iteratori e foreach
- 2 Collection, List, Set
- 3 Implementazioni di Set**

# Implementazioni di Set

## Caratteristiche dei set

- Nessun elemento duplicato (nel senso di `Object.equals()`)
- Il problema fondamentale è il metodo `contains()`, nelle soluzioni più naive (con iteratore) potrebbe applicare una ricerca sequenziale, e invece si richiedono performance migliori

## Approccio 1: HashSet

Si usa il metodo `Object.hashCode()` come funzione di **hash**, usata per posizionare gli elementi in uno store di elevate dimensioni

## Approccio 2: TreeSet

Specializzazione di `SortedSet` e di `NavigableSet`. Gli elementi sono ordinati, e quindi organizzabili in un albero (red-black tree) per avere accesso in tempo logaritmico

## Idea di base: tecnica di hashing (via `Object.hashCode()`)

- Si crea un array di elementi più grande del necessario (p.e. almeno il 25% in più), di dimensione `size`
- Aggiunta di un elemento `e`
  - ▶ lo si inserisce in posizione `e.hashCode() % size`
  - ▶ se la posizione è già occupata, lo si inserisce nella prima disponibile
  - ▶ se l'array si riempie va espanso e si deve fare il rehashing
- Ricerca di un elemento `f`
  - ▶ si guarda a partire da `f.hashCode() % size`, usando `Object.equals()`
  - ▶ La funzione di hashing deve evitare il più possibile le collisioni
- Risultato: scritture/letture sono  $O(1)$  ammortizzato

## Dettagli interni

- Realizzata tramite `HashMap`, che approfondiremo in futuro

# Costruttori di HashSet

```
1 public class HashSet<E>
2     extends AbstractSet<E>
3     implements Set<E>, Cloneable, java.io.Serializable {
4
5     // Set vuoto, usa hashmap con capacità 16
6     public HashSet() {...}
7
8     // Set con elementi di c, usa hashmap del 25% più grande di c
9     public HashSet(Collection<? extends E> c) {...}
10
11    // Set vuoto
12    public HashSet(int initialCapacity, float loadFactor) {...}
13
14    // Set vuoto, loadFactor = 0.75
15    public HashSet(int initialCapacity) {...}
16
17    /* Gli altri metodi di Collection seguono... */
18 }
```



# `equals()` e `hashCode()`

## La loro corretta implementazione è cruciale

- Le classi di libreria di Java sono già OK
- `Object` uguaglia lo stesso oggetto e l'hashing restituisce la posizione in memoria..
- .. quindi nuove classi devono ridefinire `equals()` e `hashCode()` opportunamente

## Quale funzione di hashing?

- oggetti `equals` devono avere lo stesso `hashCode`
- non è detto il viceversa, ma è opportuno per avere buone performance di `HashSet`
- si veda ad esempio:  
[`http://en.wikipedia.org/wiki/Java\_hashCode\(\)`](http://en.wikipedia.org/wiki/Java_hashCode())
- Eclipse fornisce la generazione di un `hashCode` ragionevole



# Esempio: Persona pt.1

```
1 public class Persona {
2
3     final private String nome;
4     final private int annoNascita;
5     final private boolean sposato;
6
7     public Persona(String nome, int annoNascita, boolean sposato) {
8         this.nome = nome;
9         this.annoNascita = annoNascita;
10        this.sposato = sposato;
11    }
12
13    public boolean isSposato() {
14        return this.sposato;
15    }
16
17    public String getName() {
18        return this.nome;
19    }
20
21    public String toString() {
22        return this.nome + ":" + this.annoNascita + ":" + (this.sposato ? "spos" : "non-spos"
23        );
24    }
25
26    public int hashCode() {
27        final int prime = 31;
28        int result = 1;
29        result = prime * result + annoNascita;
30        result = prime * result + ((nome == null) ? 0 : nome.hashCode());
31        result = prime * result + (sposato ? 1231 : 1237);
32        return result;
33    }
```

## Esempio: Persona pt.2

```
1 public boolean equals(Object obj) {
2     if (this == obj) {
3         return true;
4     }
5     if (obj == null) {
6         return false;
7     }
8     if (!(obj instanceof Persona)) {
9         return false;
10    }
11    Persona other = (Persona) obj;
12    if (annoNascita != other.annoNascita) {
13        return false;
14    }
15    if (nome == null) {
16        if (other.nome != null) {
17            return false;
18        }
19    } else if (!nome.equals(other.nome)) {
20        return false;
21    }
22    if (sposato != other.sposato) {
23        return false;
24    }
25    return true;
26 }
27
28
```

# UseHashSetPersona

```
1 public class UseHashSetPersona{
2     public static void main(String[] s){
3         // HashSet è un dettaglio, lavorare sempre su Set!
4         final Set<Persona> set = new HashSet<>();
5
6         // Aggiungo 4 elementi
7         set.add(new Persona("Rossi",1960,false));
8         set.add(new Persona("Bianchi",1980,true));
9         set.add(new Persona("Verdi",1972,false));
10        set.add(new Persona("Neri",1968,false));
11        System.out.println(set);
12
13        // Testo presenza/assenza di 2 elementi
14        final Persona p1 = new Persona("Rossi",1960,false);
15        final Persona p2 = new Persona("Rossi",1961,false);
16        System.out.println("Cerco "+p1+" esito "+set.contains(p1));
17        System.out.println("Cerco "+p2+" esito "+set.contains(p2));
18
19        // Iterazione: nota, fuori ordine rispetto all'inserimento
20        for (final Persona p: set){
21            System.out.println("Itero: "+p+" hash = "+p.hashCode());
22        }
23    }
24 }
```

# TreeSet<E>

## Specializzazione NavigableSet (e SortedSet)

- Assume che esista un ordine fra gli elementi
- Quindi ogni elemento ha una sua posizione nell'elenco
- Questo consente l'approccio dicotomico alla ricerca
- Consente funzioni aggiuntive, come le iterazioni in un intervallo

## Realizzazione ordinamento: due approcci (interno o esterno)

1. O con elementi che implementano direttamente Comparable
  - ▶ Nota che, p.e., Integer implementa Comparable<Integer>
2. O attraverso un Comparator esterno fornito alla `new`

## Implementazione TreeSet

- Basata su red-black tree (albero binario bilanciato)
- Tempo logaritmico per inserimento, cancellazione, e ricerca

# Comparazione “interna” agli elementi

```
1 public interface Comparable<T> {  
2     /* returns: 0 (this == o), positive (this > o)  
3         negative (this < o) */  
4     public int compareTo(T o);  
5 }
```

```
1 class Integer extends Number implements Comparable<Integer>{ ... }  
2 class String extends Object implements Comparable<String>,...{ ... }  
3 // >100 classi delle librerie di Java seguono questo approccio
```

```
1 public class Persona implements Comparable<Persona>{  
2     ...  
3     // Esempio di implementazione di compareTo:  
4     // - ordine di anno di nascita, e poi per nome..  
5     public int compareTo(Persona p){  
6         return (this.annoNascita != p.annoNascita)  
7             ? this.annoNascita - p.annoNascita  
8             : this.nome.compareTo(p.nome);  
9     }  
10 }
```

# Esempi di comparazione interna

```
1 import java.util.*;
2
3 public class UseComparison {
4
5     public static void main(String[] s) {
6
7         System.out.println("abc vs def: " + "abc".compareTo("def")); // neg
8         System.out.println("1 vs 2: " + new Integer(1).compareTo(2)); // neg
9
10        final Persona p1 = new Persona("Rossi", 1960, false);
11        final Persona p2 = new Persona("Rossi", 1972, false);
12        final Persona p3 = new Persona("Bianchi", 1972, false);
13        final Persona p4 = new Persona("Bianchi", 1972, true);
14
15        System.out.println(p1 + " vs " + p2 + ": " + p1.compareTo(p2)); // pos
16        System.out.println(p2 + " vs " + p3 + ": " + p2.compareTo(p3)); // pos
17        System.out.println(p3 + " vs " + p4 + ": " + p3.compareTo(p4)); // zero
18
19        System.out.println(new TreeSet<Integer>(Arrays.asList(4,3,2,1)));
20        // 1,2,3,4
21    }
22 }
```



# Interfacce SortedSet e NavigableSet

```
1 public interface SortedSet<E> extends Set<E> {
2     Comparator<? super E> comparator();
3     SortedSet<E> subSet(E fromElement, E toElement);
4     SortedSet<E> headSet(E toElement); // fino a toElement
5     SortedSet<E> tailSet(E fromElement); // da toElement
6     E first();
7     E last();
8 }
```

```
1 public interface NavigableSet<E> extends SortedSet<E> {
2     E lower(E e); // Elemento prima di e
3     E floor(E e); // Elemento prima di e (e incluso)
4     E ceiling(E e); // Elemento dopo e (e incluso)
5     E higher(E e); // Elemento dopo e
6     E pollFirst(); // Torna ed elimina il primo se esiste
7     E pollLast(); // Torna ed elimina l'ultimo se esiste
8     NavigableSet<E> descendingSet(); // Set con ordine invertito
9     Iterator<E> descendingIterator(); // .. e relativo iteratore
10    NavigableSet<E> subSet(E fromElement, boolean fromInclusive,
11                           E toElement, boolean toInclusive);
12    NavigableSet<E> headSet(E toElement, boolean inclusive);
13    NavigableSet<E> tailSet(E fromElement, boolean inclusive);
14 }
```

# UseTreeSetPersona: comparazione interna

```
1 public class UseTreeSetPersona {
2     public static void main(String[] s) {
3
4         final List<Integer> l = Arrays.asList(new Integer[] { 10, 20, 30, 40 });
5         // TreeSet è un dettaglio, lavorare sempre sull'interfaccia
6
7         final NavigableSet<Persona> set = new TreeSet<>();
8         set.add(new Persona("Rossi", 1960, false));
9         set.add(new Persona("Bianchi", 1980, true));
10        set.add(new Persona("Verdi", 1972, false));
11        set.add(new Persona("Neri", 1972, false));
12        set.add(new Persona("Neri", 1968, false));
13
14        // Iterazione in ordine, poi al contrario, poi fino al 1970
15        for (final Persona p : set) {
16            System.out.println("Itero: " + p + " hash = " + p.hashCode());
17        }
18        for (final Persona p : set.descendingSet()) {
19            System.out.println("Itero al contrario: " + p);
20        }
21        final Persona limit = new Persona("", 1970, false);
22        for (final Persona p : set.headSet(limit, false)) {
23            System.out.println("Itero fino al 1970: " + p);
24        }
25    }
26 }
```



# Costruttori di TreeSet, e comparatore “esterno”

```
1 public class TreeSet<E> extends AbstractSet<E>
2     implements NavigableSet<E>, Cloneable, java.io.Serializable{
3
4     // Set vuoto di elementi confrontabili
5     public TreeSet() {...}
6
7     // Set vuoto con comparatore fornito
8     public TreeSet(Comparator<? super E> comparator) {...}
9
10    // Set con gli elementi di c, confrontabili tra loro
11    public TreeSet(Collection<? extends E> c) {...}
12
13    // Set con gli elementi di c, e che usa il loro ordering
14    public TreeSet(SortedSet<E> s) {...}
15
16    /* Seguono i metodi di NavigableSet e SortedSet */
17 }
```

```
1 public interface Comparator<T> {
2     // 0 if o1 == o2, neg if o1 < o2, pos is o1 > o2
3     int compare(T o1, T o2);
4 }
```

# Definizione di un comparatore esterno

```
1 import java.util.Comparator;
2
3 /* Implementa la politica di confronto esternamente a Persona */
4
5 public class PersonaComparator implements Comparator<Persona> {
6
7     // Confronto prima sul nome, poi sull'anno
8     public int compare(Persona o1, Persona o2) {
9         return o1.getNome().equals(o2.getNome())
10             ? o1.getAnnoNascita() - o2.getAnnoNascita()
11             : o1.getNome().compareTo(o2.getNome());
12     }
13
14 }
15 }
```



## UseTreeSetPersona2: comparazione esterna

```
1 import java.util.*;
2
3 public class UseTreeSetPersona2{
4
5     public static void main(String[] s){
6
7         // TreeSet è un dettaglio, lavorare sempre sull'interfaccia
8         final Set<Persona> set =
9             new TreeSet<>(new PersonaComparator());
10
11         set.add(new Persona("Rossi",1960,false));
12         set.add(new Persona("Bianchi",1980,true));
13         set.add(new Persona("Verdi",1972,false));
14         set.add(new Persona("Neri",1972,false));
15         set.add(new Persona("Neri",1968,false));
16
17         // Iterazione in ordine
18         for (final Persona p: set){
19             System.out.println(p);
20         }
21     }
22 }
```



## Perché il tipo `Comparator<? super E>`

Data una classe `SortedSet<E>` il suo comparatore ha tipo `Comparator<? super E>`, perché non semplicemente `Comparator<E>`?

### È corretto

- `Comparator` ha metodi che hanno `E` solo come argomento
- quindi l'uso di `Comparator<? super E>` è una generalizzazione di `Comparator<E>`

### È utile

- Supponiamo di aver costruito un comparatore per `SimpleLamp`, e che questo sia usabile anche per tutte le specializzazioni successivamente costruite (è la situazione tipica)
- Anche un `SortedSet<UnlimitedLamp>` deve poter usare il `Comparator<SimpleLamp>`, ma questo è possibile solo grazie al suo tipo atteso `Comparator<? super E>`

# Un esempio di design con le collezioni

Implementare questa interfaccia che modella un archivio persone

```
1 import java.util.Set;
2
3 public interface Archive {
4
5     void add(String nome, int annoNascita, boolean sposato);
6
7     void remove(String nome, int annoNascita);
8
9     int size();
10
11     Set<String> allMarried();
12 }
```



# Una soluzione con HashSet

```
1 public class ArchiveImpl implements Archive {
2
3     private final Set<Persona> set = new HashSet<>();
4
5     public void add(String nome, int annoNascita, boolean sposato) {
6         this.set.add(new Persona(nome, annoNascita, sposato));
7     }
8
9     public void remove(String nome, int annoNascita) {
10        this.set.remove(new Persona(nome, annoNascita, false));
11    }
12
13    public int size() {
14        return this.set.size();
15    }
16
17    public Set<String> allMarried() {
18        final Set<String> newset = new HashSet<>();
19        for (final Persona p: this.set) {
20            if (p.isSposato()) {
21                newset.add(p.getName());
22            }
23        }
24        return newset;
25    }
26 }
```

# Scenario d'uso dell'archivio

```
1 public class UseArchive {
2
3     public static void main(String[] args) {
4         final Archive arc = new ArchiveImpl();
5         arc.add("Rossi", 1960, false);
6         arc.add("Bianchi", 1980, true);
7         arc.add("Verdi", 1972, true);
8         arc.add("Neri", 1968, false);
9         arc.remove("Neri", 1968);
10
11         System.out.println(arc.size()); // 3
12         System.out.println(arc.allMarried()); // [Bianchi,Verdi]
13     }
14 }
15 }
```

