

# Programmazione Orientata agli Oggetti

---

Collezioni  
Liste generiche

# Sommario

- Introduzione alle Collezioni
  - Interface Collection<E>
  - Iterare una collezione: Iterator<E>
  - Rimuovere elementi da una collezione
  - Boxing-unboxing
- Liste Generiche
  - aggiungere elementi
  - iterare sugli elementi della lista
- Ordinamento di liste
  - Comparable, Comparator

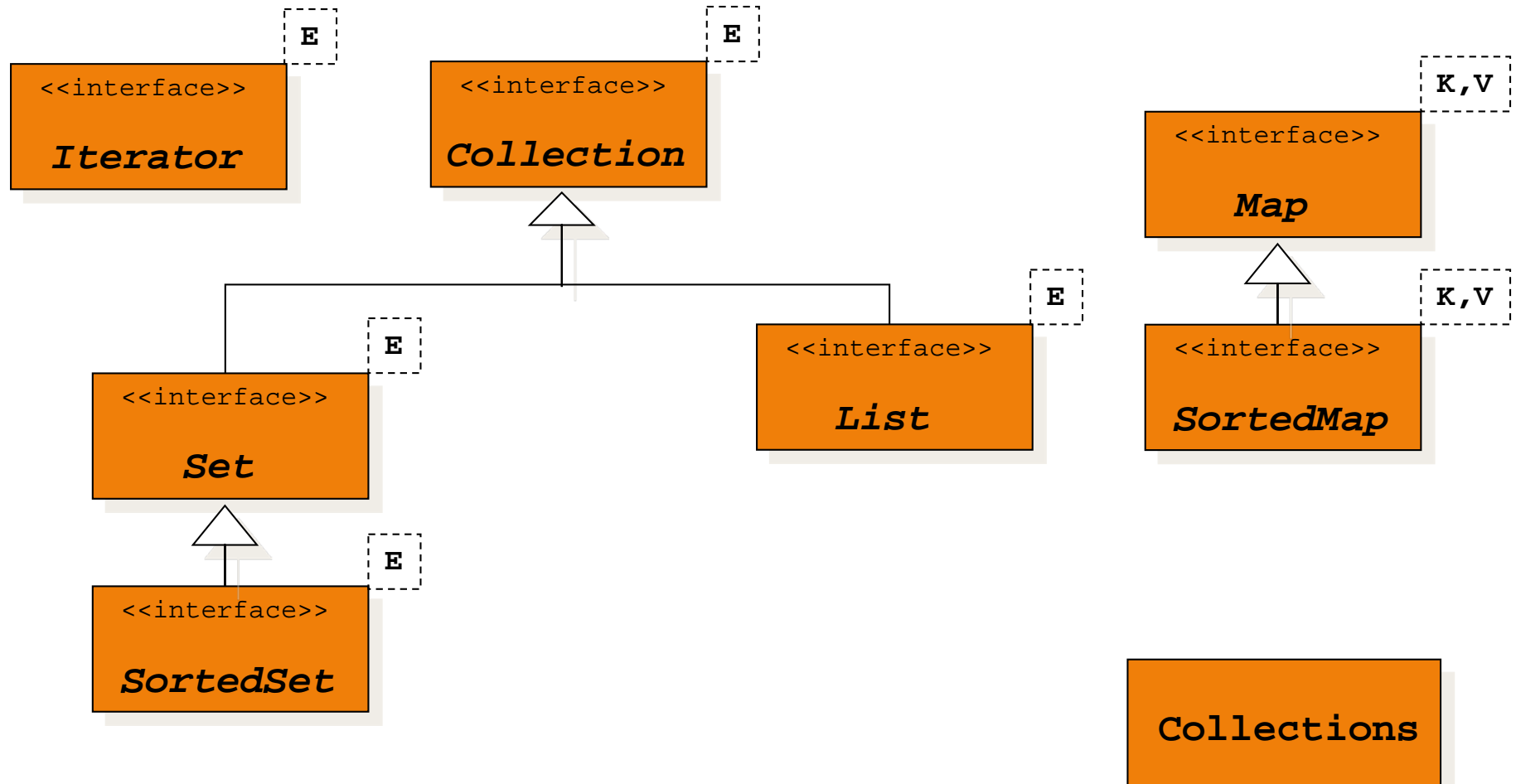
# Introduzione

- Molte applicazioni richiedono di gestire collezioni di oggetti
- Gli **array** sono uno strumento di basso livello
  - La dimensione di una collezione in genere non è nota a priori e può variare notevolmente
  - Negli esercizi fatti fino ad ora abbiamo supposto un numero massimo di elementi, e abbiamo usato un indicatore di riempimento per tenere traccia del numero di elementi memorizzati nell'array
  - Possiamo avere bisogno di molte modalità di accesso (non solo indicizzato; ad es. LIFO, FIFO, ecc. ecc.)
  - Ci può essere la necessità di mantenere gli elementi ordinati

# Le collezioni del package `java.util`

- Nella libreria di base di Java abbiamo un package che ci offre un vasto insieme di interfacce e di classi per la gestione di collezioni di oggetti
- Il package ha subito una sostanziale modifica in seguito alla introduzione dei Generics

# Collezioni



# Un primo sguardo: `Collection<E>`

- L'interface `Collection<E>` dichiara i metodi di una generica collezione
- Questi metodi permettono di svolgere operazioni quali:
  - Aggiungere un elemento alla collezione
  - Verificare la dimensione della collezione
  - Verificare se la collezione è vuota
  - Aggiungere tutti gli elementi di un'altra collezione
  - Ottenere un *iteratore* con cui scandire la collezione

# Uno primo sguardo: `Set<E>`

- L'interface `Set<E>` estende `Collection<E>`:  
è una collezione che non può contenere *duplicati*
- Offre tutti e soli i metodi della interface `Collection`, con la restrizione che le classi che la implementano si impegnano a non ammettere la presenza di elementi *duplicati*
  - Per poter parlare di duplicati sarà necessario stabilire e modellare un criterio di equivalenza tra gli elementi dell'insieme

# Uno sguardo d'insieme: `List<E>`

- L'interface `List<E>` estende `Collection<E>` e corrisponde ad una sequenza, ovvero una collezione ordinata di elementi
- Le liste, rispetto agli insiemi, possono contenere elementi duplicati
- Oltre alle operazioni offerte dal supertipo `Collection`, la interface `List` include altre operazioni specifiche, quali:
  - Accesso posizionale: permette di accedere agli elementi in base alla loro posizione nella lista (in maniera simile a quanto avviene per gli array)
  - Ricerca: permette di ricercare un elemento nella lista e ritorna la sua posizione all'interno della sequenza



# Uno sguardo d'insieme: `Map<K, V>`

- L'interface `Map<K, V>` offre le operazioni di una mappa, o dizionario: una mappa è una collezione di coppie chiave-valore
- L'interface `Map<K, V>` dichiara i metodi per operazioni quali:
  - Ottenere il valore associato ad una chiave
  - Cancellare una coppia in cui compare una chiave
  - Inserire una nuova coppia nella mappa
  - Ottenere una collezione contenente tutte le chiavi o tutti i valori

# Uno sguardo d'insieme: Collections

- La classe `java.util.Collections` (al plurale: attenzione alla s finale!)
  - offre un vasto insieme di metodi (statici) generici che implementano utili algoritmi per la manipolazione di liste quali:
    - ordinamento
    - ricerca max e min
    - shuffle
    - ...

# Sommario

- Introduzione alle Collezioni
  - Interface Collection<E>
  - Iterare una collezione: Iterator<E>
  - Rimuovere elementi da una collezione
  - Boxing-unboxing
- Liste Generiche
  - aggiungere elementi
  - iterare sugli elementi della lista
- Ordinamento di liste
  - Comparable, Comparator

# Interface `Collection<E>`

- L'interface `Collection<E>` dichiara i metodi di una collezione generica
- Questi metodi permettono di svolgere operazioni quali:
  - Aggiungere un elemento alla collezione
  - Verificare la dimensione della collezione
  - Verificare se la collezione è vuota
  - Aggiungere tutti gli elementi di un'altra collezione
  - Ottenere un *iteratore* con cui scandire la collezione

# Interface Collection<E>

```
public interface Collection<E> extends Iterable<E> {  
    //Basic operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element); //optional  
    boolean remove(Object element); //optional  
    Iterator<E> iterator();  
    //Bulk operations  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c); //optional  
    boolean removeAll(Collection<?> c); //optional  
    boolean retainAll(Collection<?> c); //optional  
    void clear(); //optional  
    //Array operations  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
}
```

# Collection<E>: metodi base

- **int size();**  
ritorna il numero di elementi presenti nella collezione
- **boolean isEmpty();**  
ritorna **true** se la collezione è vuota
- **boolean contains(Object element);**  
ritorna **true** se la collezione contiene un elemento uguale a quello passato come parametro (l'uguaglianza è verificata dal metodo **equals()**)
- **boolean add(E element);**  
aggiunge alla collezione l'elemento passato; ritorna **true** se la collezione è cambiata dopo la chiamata a questo metodo
- **boolean remove(Object element);**  
rimuove dalla collezione gli elementi uguali all'oggetto passato come parametro (l'uguaglianza è verificata dal metodo **equals()**). Ritorna **true** se la collezione è cambiata dopo l'invocazione del metodo
- **Iterator<E> iterator();**  
restituisce un oggetto **Iterator**, per iterare sugli elementi della collezione

# Collection<E>: metodi bulk

- **boolean containsAll(Collection<?> c);**  
ritorna **true** se la collezione contiene tutti gli elementi della collezione passata come parametro
- **boolean addAll(Collection<? extends E> c);**  
aggiunge alla collezione tutti gli elementi della collezione passata come parametro; ritorna **true** se la collezione è cambiata dopo l'invocazione di questo metodo
- **boolean removeAll(Collection<?> c);**  
rimuove dalla collezione tutti gli elementi uguali (l'uguaglianza è verificata dal metodo **equals()**) che sono contenuti nella collezione passata come parametro; ritorna **true** se la collezione è cambiata dopo l'invocazione di questo metodo
- **boolean retainAll(Collection<?> c);**  
rimuove dalla collezione tutti gli elementi che non sono presenti nella collezione passata come parametro; ritorna **true** se la collezione è cambiata dopo l'invocazione di questo metodo
- **void clear();**  
rimuove tutti gli elementi dalla collezione

# Sommario

- Introduzione alle Collezioni
- Interface Collection<E>
- Iterare una collezione: Iterator<E>
- Rimuovere elementi da una collezione
- Boxing-unboxing
- Liste Generiche
- Ordinamento di liste
  - Comparable, Comparator



# Iterazione: interface `Iterator<E>`

- L'iterazione di una collezione avviene attraverso un oggetto che ha la responsabilità di governare l'iterazione
- Questo oggetto, che viene chiesto alla collezione mediante il metodo `iterator()`, implementa l'interface `Iterator<E>`, che offre i metodi
  - `boolean hasNext()`
  - `E next()`
  - `void remove()`

# Iterator<E>: metodi

- **boolean hasNext( ) ;**

ritorna **true** se e solo se esiste un altro elemento da scandire

- **E next( ) ;**

restituisce il prossimo elemento della collezione nella scansione corrente ed avanza

- **void remove( ) ;**

rimuove dalla collezione l'ultimo elemento che è stato restituito dalla chiamata di **next( )**

# Iterator<E>: iterazione

- La chiamata ripetuta di **next ( )** permette di scorrere gli elementi della collezione uno alla volta
- Se si raggiunge la fine della collezione viene sollevata una eccezione (che interrompe il programma)  
**java.util.NoSuchElementException**
- Per evitare questa situazione, prima di chiamare **next ( )** si usa il metodo **hasNext ( )**, che ritorna **true** se e solo se esiste un altro elemento su cui iterare

# Sommario

- Introduzione alle Collezioni
  - Interface Collection<E>
  - Iterare una collezione: Iterator<E>
  - Rimuovere elementi da una collezione
  - Boxing-unboxing
- Liste Generiche
  - aggiungere elementi
  - iterare sugli elementi della lista
- Ordinamento di liste
  - Comparable, Comparator

# Rimuovere elementi da una collezione

- Ci sono diversi modi per rimuovere un elemento da una collezione, ciascuno è dettato da esigenze specifiche:
  - per la rimozione di un elemento uguale ad un elemento dato (passato come parametro) si usa il metodo **`remove(Object o)`** di **`Collection`**
  - per la rimozione di un elemento durante la scansione di una lista si usa il metodo **`remove()`** di **`Iterator`**

# Il metodo `remove(Object o)` di `Collection`

- boolean **remove**(Object o)
  - Removes a single instance of the specified element from this collection, if it is present (optional operation). More formally, removes an element *e* such that  $(o == null ? e == null : o.equals(e))$ , if this collection contains one or more such elements. Returns true if this collection contained the specified element (or equivalently, if this collection changed as a result of the call).
  - **Parameters:**
    - o - element to be removed from this collection, if present
  - **Returns:**
    - true if an element was removed as a result of this call

*(dalla documentazione)*

# Il metodo `remove()` di `Iterator`

- `void remove()`
  - Removes from the underlying collection the last element returned by the iterator (optional operation). This method can be called only once per call to `next`. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

*(dalla documentazione)*

# Il metodo `remove()` di `Iterator`

- Il metodo `remove()` rimuove l'elemento restituito dall'ultima chiamata di `next()`
- Non è ammesso chiamare `remove()` se prima non si è chiamato `next()`
- Es. voglio eliminare due elementi consecutivi:  
`it.remove();`  
`it.remove();`     `// ERRORE`

devo prima chiamare `next()`:

```
it.remove();  
it.next();  
it.remove();    // OK
```



# Rimuovere elementi da una collezione

- Attenzione: è un errore cercare di rimuovere elementi da una collezione con il metodo
  - `boolean remove(Object o)`di `Collection` mentre si sta visitando la collezione con un iteratore
  - la collezione verrebbe modificata "sotto i piedi" dell'iteratore
- Se si stanno cercando elementi da rimuovere attraverso un iteratore, deve essere usato il metodo `remove()` dell'iteratore

# Sommario

- Introduzione alle Collezioni
  - Interface Collection<E>
  - Iterare una collezione: Iterator<E>
  - Rimuovere elementi da una collezione
  - Boxing-unboxing
- Liste Generiche
  - aggiungere elementi
  - iterare sugli elementi della lista
- Ordinamento di liste
  - Comparable, Comparator

# Collezioni e tipi primitivi

- Nelle collezioni non si possono memorizzare tipi primitivi
- Se si vogliono gestire tipi primitivi è necessario usare le classi wrapper
- Esempio: una collezione di interi  
**`Collection<Integer> c;`**
- Dalla versione 1.5 di Java, la gestione di oggetti wrapper è semplificata dalle funzionalità di *boxing* e *unboxing*

# Boxing

- *Boxing*: è possibile assegnare direttamente tipi primitivi a oggetti wrapper
- Le seguenti istruzioni sono equivalenti:

```
int i = 0;  
Integer iWrap;  
iWrap = i;  
iWrap = 5;
```

```
int i = 0;  
Integer iWrap;  
iWrap = new Integer(i);  
iWrap = new Integer(5);
```

- È il compilatore che inserisce le istruzioni per gestire il wrapping

# Unboxing

- *Unboxing*: è possibile assegnare direttamente oggetti wrapper a tipi primitivi
- Le seguenti istruzioni sono equivalenti:

```
int i = 0;  
Integer iWrap;  
iWrap = 5;  
i = iWrap;
```

```
int i = 0;  
Integer iWrap;  
iWrap = new Integer(5);  
i = iWrap.intValue();
```

- È il compilatore che inserisce le chiamate a costruttori e metodi

# Boxing, unboxing e collezioni

- Grazie a boxing e unboxing, anche la gestione collezioni che memorizzano informazioni riconducibili a tipi primitivi è semplificata
- Le seguenti operazioni sono lecite (grazie a boxing e unboxing):

```
Collection<Integer> c;  
c = new LinkedList<Integer>( );  
int i = 4;  
c.add(i);  
c.add(5);
```

# Attenzione allo zucchero sintattico

- Le nuove versioni del compilatore tendono a semplificare la gestione dei tipi primitivi
- Tuttavia, è necessario comprendere a fondo
  - la differenza tra il concetto di tipo primitivo e la loro controparte ad oggetti, i wrapper
  - quali operazioni non sono necessarie solo grazie ai servizi offerti dalle ultime versioni del compilatore java (sarebbero necessarie con versioni precedenti)
  - quali operazioni il compilatore inserisce per conto nostro
- Perché conviene avere queste competenze?
  - per stimare meglio il numero di oggetti creati dalle nostre applicazioni
  - per migliorare la nostra capacità di ricerca delle origine degli errori sia a tempo di compilazione che di esecuzione
  - per riuscire ad usare versioni precedenti del compilatore

# Sommario

- Introduzione alle Collezioni
  - Interface Collection<E>
  - Iterare una collezione: Iterator<E>
  - Rimuovere elementi da una collezione
  - Boxing-unboxing
- Liste Generiche
  - aggiungere elementi
  - iterare sugli elementi della lista
- Ordinamento di liste
  - Comparable, Comparator



# Liste: interface `List<E>`

- Una lista è una collezione che mantiene gli elementi ordinati secondo l'ordine di inserimento (il primo elemento aggiunto alla lista, è in prima posizione, il secondo in seconda posizione, ..., l'ultimo elemento aggiunto è in ultima posizione)

# Liste: interface `List<E>`

- L'interface `List<E>` estende l'interface `Collection<E>`
- Oltre ai metodi della interface `Collection<E>`, `List<E>` offre metodi che consentono accesso e inserimento indicizzati degli elementi. Ad esempio
  - *`E` `get`(int index): Returns the element at the specified position in this list*
  - *int `indexOf`(`Object` o): Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.*

# Implementazioni di `List<E>`

- Il package `java.util` offre due diverse implementazioni di `List<E>`
  - `ArrayList<E>`
  - `LinkedList<E>`

# Implementazioni di `List<E>`: intuizione

- Diamo una intuizione della implementazione delle due classi implementano la interface `List<E>`
  - `ArrayList<E>`
  - `LinkedList<E>`
- E qualche (grossolana) indicazione su come scegliere l'implementazione più opportuna
  - NOTA: Questi aspetti sono stati approfonditi nel corso *"Algoritmi e Strutture Dati"*

# ArrayList<E>: implementazione

- Gli elementi sono memorizzati in un contenitore implementato con array con indicatore di riempimento
- Al momento della creazione, la dimensione dell'array (capacità della collezione) è inizializzata ad un valore prestabilito (ci sono costruttori e metodi che permettono di agire su questo valore)
- Quando il numero di elementi è prossimo alla capacità dell'array, viene istanziato un nuovo array di dimensione maggiore (ad esempio doppia) nel quale vengono copiati tutti gli elementi dell'array originario. Il nuovo array diventa il contenitore
- NOTA: Questi aspetti sono stati approfonditi nel corso *"Algoritmi e Strutture Dati"*

# LinkedList<E>: implementazione

- Gli elementi sono memorizzati in una lista concatenata
- Ogni elemento della lista contiene
  - un riferimento all'elemento successivo
  - un riferimento all'oggetto memorizzato
- Non è necessario stabilire una capacità iniziale
- NOTA: Questi aspetti sono stati approfonditi nel corso *"Algoritmi e Strutture Dati"*

# LinkedList<E> o ArrayList<E>?

- Molto schematicamente
  - **ArrayList<E>** conviene se:
    - La dimensione è abbastanza stabile
    - È necessario un accesso indicizzato (la classe ArrayList offre un metodo opportuno)
  - **LinkedList<E>** conviene se:
    - La dimensione può variare anche significativamente
    - Gli accessi sono perlopiù sequenziali
- NOTA: Questi aspetti sono stati approfonditi nel corso *"Algoritmi e Strutture Dati"*

# Implementazioni di `List<E>` : costruttori

- I costruttori sono sovraccarichi. In particolare facciamo osservare che esiste un costruttore che permette la creazione di una lista a partire da una collezione
- Costruttori di **`ArrayList<E>`**
  - **`ArrayList()`** *Constructs an empty list with an initial capacity of ten.*
  - **`ArrayList(Collection<? extends E> c)`** *Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.*
  - **`ArrayList(int initialCapacity)`** *Constructs an empty list with the specified initial capacity.*
- Costruttori di **`LinkedList<E>`**
  - **`LinkedList<E>`** *Constructs an empty list.*
  - **`LinkedList<E>(Collection<? extends E> c)`** *Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.*



# List<E> : Esercizio

- La classe **ArrayList<E>** implementa l'interfaccia **List<E>** (e quindi **Collection<E>**)
- Proviamo a rivedere il codice della classe **Borsa** nello studio di caso:
  - anziché usare un array per memorizzare l'insieme di attrezzi, usiamo un **ArrayList<E>**
- Vediamo come gestiamo
  - Aggiunta di un elemento
  - Scansione della lista

# List<E>: Esercizio

## Aggiungere elementi

```
public class Borsa {
    private Attrezzo[] attrezzi;
    private int numeroAttrezzi;

    public Borsa() {
        this.numeroAttrezzi = 0;
        this.attrezzi = new Attrezzo[10];
    }

    public void addAttrezzo(Attrezzo attrezzo){
        this.attrezzi[this.numeroAttrezzi] = attrezzo;
        this.numeroAttrezzi++;
    }
    ...
}
```

*Con array*

```
import java.util.List;
import java.util.ArrayList;

public class Borsa {
    private List<Attrezzo> attrezzi;

    public Borsa() {
        this.attrezzi = new ArrayList<Attrezzo>();
    }

    public boolean addAttrezzo(Attrezzo attrezzo){
        return this.attrezzi.add(attrezzo);
    }
    ...
}
```

*Con ArrayList*

# List<E>: Esercizio

## Osservazioni

- Dobbiamo importare:  
`java.util.List`  
`java.util.ArrayList`
- Non ci dobbiamo preoccupare di stabilire a priori le dimensioni massime della collezione
- Non ci dobbiamo preoccupare di gestire l'indicatore di riempimento, che memorizza il numero di elementi effettivamente memorizzati nell'array

# List<E>: Esercizio

## Aggiungere elementi

- L'aggiunta di elementi in un **ArrayList<E>** viene realizzata tramite il metodo **add(E e1)**
- Questo metodo aggiunge un riferimento ad oggetto (istanza di tipo **E**) nell'ultima posizione della collezione
- Gli elementi della lista rimangono ordinati secondo l'ordine di inserimento
  - L'oggetto inserito per primo è nella prima posizione, l'oggetto inserito per secondo è nella seconda posizione, ..., l'oggetto inserito per ultimo è in ultima posizione

# List<E>: Esercizio

## Osservazioni

- La lista aumenta la sua capacità se necessario
- Mantiene un conteggio del numero di elementi (possiamo accedere a questo valore con il metodo `int size()`)
- Mantiene gli oggetti in ordine di inserimento
- I dettagli di come tutto ciò viene realizzato ci è nascosto
  - È importante? Non conoscere questi dettagli ci impedisce di usare la collezione?

# List<E>: Esercizio

## Scandire la lista con un iteratore

```
public class Borsa {
    private Attrezzo[] attrezzi;
    private int numeroAttrezzi;
    ...
    public int getPeso(){
        int pesoTotale = 0;
        for(int i=0; i<this.numeroAttrezzi; i++) {
            Attrezzo a;
            a = this.attrezzi[i];
            pesoTotale += a.getPeso();
        }
        return pesoTotale;
    }
    ...
}
```

*Con array*

```
import java.util.List;
import java.util.ArrayList;
```

```
public class Borsa {
    private List<Attrezzo> attrezzi;
    ...
    public int getPeso(){
        int pesoTotale = 0;
        Iterator<Attrezzo> iteratore = this.attrezzi.iterator();
        while (iteratore.hasNext()) {
            Attrezzo a;
            a = iteratore.next();
            pesoTotale += a.getPeso();
        }
        return pesoTotale;
    }
    ...
}
```

*Con ArrayList*

# List<E>: Esercizio

## Scandire la lista con un iteratore

- Commentiamo il codice

- `Iterator<Attrezzo> iteratore =  
this.attrezzi.iterator();`

Abbiamo chiesto alla lista di darci un oggetto che sa gestire l'iterazione su una collezione di oggetti Attrezzo

- `while (iteratore.hasNext())`

Fintanto che ci sono ancora elementi da scandire: questo ce lo dice l'oggetto iteratore

- `a = iteratore.next();`

Attraverso il metodo `next()` ci facciamo dare dall'iteratore il prossimo elemento della scansione

# Rimuovere elementi da una collezione

- Ci sono diversi modi per rimuovere un elemento da una collezione, ciascuno è dettato da esigenze specifiche:
  - per la rimozione di un elemento uguale (secondo il metodo `equals`) ad un elemento dato (passato come parametro) si usa il metodo `remove(Object o)` di `Collection`
  - per la rimozione di un elemento durante la scansione di una lista si usa il metodo `remove()` di `Iterator`



# Il metodo `remove()` di `Iterator`

- `void remove()`
  - Removes from the underlying collection the last element returned by the iterator (optional operation). This method can be called only once per call to `next`. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

*(dalla documentazione)*

# List<E>: Esercizio

## Rimuovere un elemento dalla lista

```
import java.util.List;
import java.util.ArrayList;

public class Borsa {
    private List<Attrezzi> attrezzi;
    ...
    public Attrezzo removeAttrezzo(String nomeAttrezzo) {
        Attrezzo a = null;
        Iterator<Attrezzo> iteratore = this.attrezzi.iterator();
        while (iteratore.hasNext()) {
            a = iteratore.next();
            if (a.getNome().equals(nomeAttrezzo)) {
                iteratore.remove();
                return a;
            }
        }
        return null;
    }
    ...
}
```

*Con ArrayList*

# Esercizio: la semantica di `Iterator`

- Per comprendere la semantica dei metodi di una classe non esiste metodo più preciso di una scriverci sopra una batteria di test
- Per **`Iterator<E>`** e **`List<E>`**
  - scriviamo test per i tre metodi dell'interfaccia **`Iterator<E>`**

# Test per capire la semantica

```
import ...junit framework import omessi...
import java.util.*;
public class TestIterator {
    private List<String> emptyList;
    private List<String> oneElementList;
    private String singolo;
    @Before
    public void setUp() {
        this.emptyList = new ArrayList<String>();
        this.oneElementList = new ArrayList<String>();
        this.singolo = new String("singolo");
        this.oneElementList.add(this.singolo);
    }
    @Test public void testNext() {...}
    @Test public void testHasNext() {...}
    @Test public void testRemove() {...}
}
```

# Test di `Iterator.hasNext()`

@Test

```
public void testHasNext_noNext() {  
    Iterator<String> it =  
        this.emptyList.iterator();  
    assertNotNull(it);  
    assertFalse(it.hasNext());  
}
```

@Test

```
public void testHasNext_yesNext() {  
    Iterator<String> it =  
        this.oneElementList.iterator();  
    assertNotNull(it);  
    assertTrue(it.hasNext());  
    it.next();  
    assertFalse(it.hasNext());  
}
```

# Test di `Iterator.next()`

@Test

```
public void testNext_oneElement() {  
    Iterator<String> it =  
        this.oneElementList.iterator();  
    assertNotNull(it);  
    assertTrue(it.hasNext());  
    String elemento = it.next();  
    assertEquals(this.singolo, elemento);  
}
```

# Test di `Iterator.next()`

@Test

```
public void testNext_twoElements() {  
    List<String> twoElementList =  
        new ArrayList<String>();  
    twoElementList.add(new String("primo"));  
    twoElementList.add(new String("secondo"));  
    Iterator<String> it =  
        twoElementList.iterator();  
    assertNotNull(it);  
    assertTrue(it.hasNext());  
    assertEquals("primo", it.next());  
    assertTrue(it.hasNext());  
    assertEquals("secondo", it.next());  
    assertFalse(it.hasNext());  
}
```

# Test di `Iterator.remove()`

@Test

```
public void testRemove() {  
    Iterator<String> it =  
        this.oneElementList.iterator();  
    assertNotNull(it);  
    assertTrue(it.hasNext());  
    String elemento = it.next();  
    assertFalse(this.oneElementList.isEmpty());  
    it.remove();  
    assertTrue(this.oneElementList.isEmpty());  
}
```



# Iterazione: for-each

- Per iterare su **tutti** gli elementi di una collezione (e quindi anche di una lista) è possibile usare la forma "for-each" dell'istruzione **for**

**for**( *Tipo elemento : collezione*)  
    *istruzione\_su elemento*

# List<E>: Esercizio

Scandire la lista con l'istruzione for-each

```
import java.util.List;
import java.util.ArrayList;

public class Borsa {
    private List<Attrezzo> attrezzi;
    ...
    public int getPeso(){
        int pesoTotale = 0;
        for(Attrezzo a : this.attrezzi)
            pesoTotale += a.getPeso();
        return pesoTotale;
    }
    ...
}
```

*Con ArrayList  
e istruzione for-each*

# Esercizio (a casa)

- Compilare ed eseguire i test riportati nelle trasparenze precedenti
- Analizzare, compilare ed eseguire la classe di test ListTest riportata nella prossima trasparenza
  - aggiungere opportuni metodi di test per verificare la semantica dei metodi:  
retainAll(Collection<?> c), contains(Object o),  
containsAll(Collection<?> c)
  - Suggerimento:

# Esercizio (a casa)

```
public class ListTest {
    private Collection<Integer> c;
    private Collection<Integer> t;

    @before
    public void setUp () {
        c = new LinkedList<Integer>();
        t = new ArrayList<Integer>();
        c.add(1);
        c.add(2);
        c.add(3);
        t.add(1);
        t.add(2);
    }

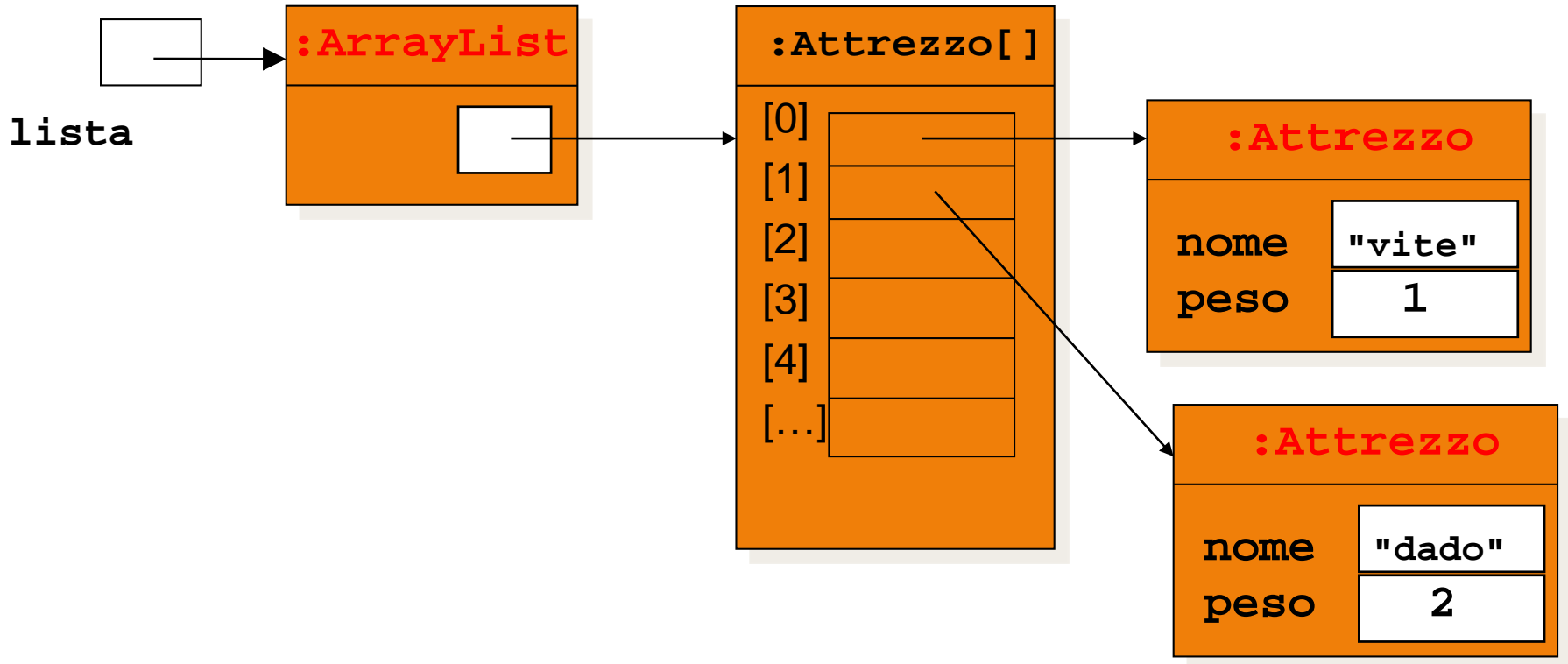
    @Test
    public void testRemoveAll() {
        assertTrue(c.removeAll(t));
        Iterator<Integer> it = c.iterator();
        assertTrue(it.hasNext());
        assertEquals(3,it.next().intValue());
        assertFalse(it.hasNext());
    }
}
```

# Liste: diagramma degli oggetti

- Nel seguito introduciamo una notazione grafica per la rappresentazione di oggetti **ArrayList** e **LinkedList**
  - la rappresentazione proposta è una astrazione (molto semplificata, ma utile a fini didattici) della rappresentazione interna delle due implementazioni

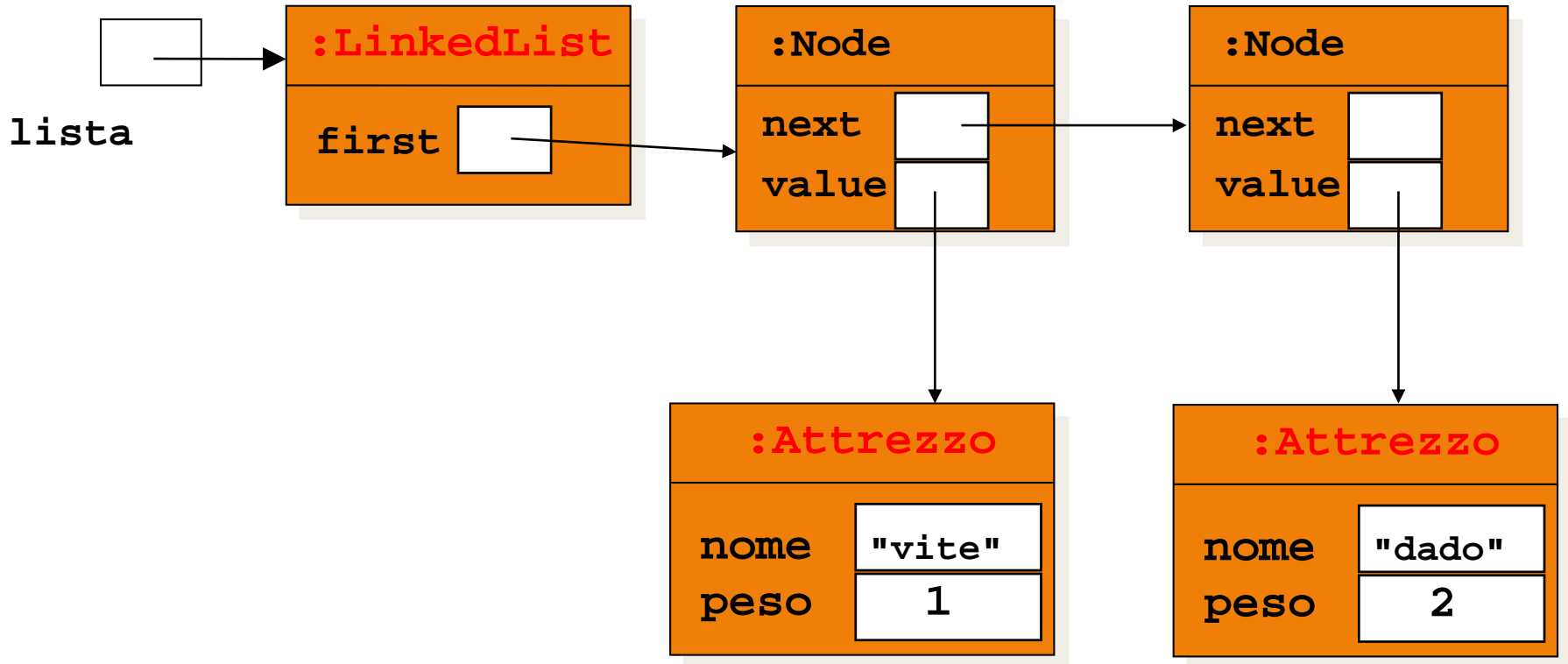
# ArrayList: Diagramma degli oggetti

```
List<Attrezzo> lista;  
lista = new ArrayList<Attrezzo>;  
lista.add(new Attrezzo("vite",1);  
lista.add(new Attrezzo("dado",2);
```



# LinkedList: Diagramma degli oggetti

```
List<Attrezzo> lista;  
lista = new LinkedList<Attrezzo>;  
lista.add(new Attrezzo("vite",1);  
lista.add(new Attrezzo("dado",2);
```



# Sommario

- Introduzione alle Collezioni
  - Interface Collection<E>
  - Iterare una collezione: Iterator<E>
  - Rimuovere elementi da una collezione
  - Boxing-unboxing
- Liste Generiche
  - aggiungere elementi
  - iterare sugli elementi della lista
- Ordinamento di liste
  - Comparable, Comparator



# Ordinamenti e ricerche

- Abbiamo metodi che implementano algoritmi efficienti per
  - ordinare una lista
  - ricercare la posizione di un elemento in una lista ordinata
  - ricercare l'elemento "più grande"/"più piccolo" in una lista

# Ordinamenti e ricerche

- Queste operazioni hanno un senso se esiste una relazione d'ordine tra gli elementi della lista
  - in altri termini, gli elementi della lista devono sapersi confrontare
  - oppure ci deve essere un oggetto esterno che sa come confrontare due oggetti della lista

# Definire un criterio di ordinamento

- I metodi (per le operazioni di ordinamento e ricerca) della classe **Collections** si affidano all'esistenza di un criterio di ordinamento specifico per il tipo degli elementi contenuti nella collezione (o suo supertipo)
- La responsabilità di modellare il criterio di ordinamento può essere affidata, in alternativa:
  - alla stessa classe degli oggetti contenuti, che deve implementare una apposita interfaccia **java.lang.Comparable**
  - ad una classe esterna alla classe degli oggetti contenuti; tale classe esiste solo con l'obiettivo di confrontarli, si chiama *comparatore* e rispetta l'interfaccia **java.util.Comparator**

# L'interface `java.lang.Comparable<T>`

- L'interface `java.lang.Comparable<T>` ha un solo metodo:

```
public int compareTo(T that)
```

che deve restituire un valore che è:

- minore, uguale, maggiore di zero  
a seconda che l'oggetto corrente sia
- minore, uguale, maggiore dell'oggetto riferito dal  
parametro `that`

# L'interface `java.lang.Comparable<T>`

- Molte importanti classi della libreria standard implementano `java.lang.Comparable<T>`, es.
  - `java.lang.String`
  - `java.util.Calendar`
  - `java.util.Date`
  - `java.io.File`
  - `java.net.URI`
  - tutte le classi wrapper
  - ... e molte altre ancora

# L'interface `java.lang.Comparable<T>`: esempio

```
public class Persona implements Comparable<Persona> {  
    private String nome;  
    private int eta;  
  
    public Persona(String nome, int eta) {  
        this.nome = nome;  
        this.eta = eta;  
    }  
  
    public String getNome() {  
        return this.nome;  
    }  
  
    public int getEta() {  
        return this.eta;  
    }  
  
    public int compareTo(Persona p) {  
        return this.nome.compareTo(p.getNome());  
    }  
}
```

# L'interface `java.lang.Comparable<T>`: esempio

```
public class PersonaTest {
    @Test
    public void testCompareTo() {
        Persona p1 = new Persona("Paolo", 10);
        Persona p2 = new Persona("Valter", 5);

        assertTrue(p1.compareTo(p2) < 0);    // <0

        Persona p3 = new Persona("Paolo", 10);
        assertTrue (p1.compareTo(p3) == 0); // 0

        Persona p4 = new Persona("Anna", 8);
        assertTrue (p1.compareTo(p4) > 0);   // >0
    }
}
```

# Ordinamento "naturale"

- Su un oggetto **List<T>** contenente oggetti che implementano l'interfaccia **java.lang.Comparable<T>**
  - si possono effettuare ricerche
  - si può calcolare il massimo e il minimo
  - si può effettuare l'ordinamento
- Queste operazioni si basano sull'ordinamento "naturale", ovvero sulla relazione d'ordine implementata dal metodo **compareTo( )**



# Ordinare una lista

- Una lista **List<T>** i cui elementi implementino l'interface **Comparable<T>** può essere ordinata (secondo l'ordinamento naturale) mediante il metodo statico **Collections.sort()**
  - NOTA: se gli elementi della lista **List<T>** non implementano l'interface **java.lang.Comparable<T>** si solleva un errore a tempo di compilazione

# Ordinare una lista secondo l'ordinamento naturale

```
public class SortTest {  
    @Test  
    public void testSort() {  
        List<Persona> l = new LinkedList<Persona>();  
        l.add(new Persona("Valter", 5));  
        l.add(new Persona("Paolo", 10));  
        l.add(new Persona("Giacomo", 7));  
        l.add(new Persona("Alessandro", 8));  
        Collections.sort(l);  
        assertEquals("Alessandro", l.get(0).getNome());  
        assertEquals("Giacomo", l.get(1).getNome());  
        assertEquals("Paolo", l.get(2).getNome());  
        assertEquals("Valter", l.get(3).getNome());  
    }  
}
```

NOTA: se **Persona** non implementasse **Comparable<Persona>**, si sollevarebbe un errore a tempo di compilazione

# Ottenere l'elemento max/min di una lista

- Da una lista **List<T>** i cui elementi implementino l'interface **java.lang.Comparable<T>** può essere ottenuto l'elemento massimo/minimo (rispetto all'ordinamento naturale) mediante il metodo statico **Collections.max()** / **Collections.min()**
  - NOTA: se gli elementi della lista **List<T>** non implementano l'interface **java.lang.Comparable<T>** si solleva un errore a tempo di compilazione

# Ottenere l'elemento max/min di una lista

```
public class SortTest {  
    @Test  
    public void testSort() {  
        List<Persona> l = new LinkedList<Persona>();  
        l.add(new Persona("Valter"), 5);  
        l.add(new Persona("Paolo"), 10);  
        l.add(new Persona("Giacomo"), 7);  
        l.add(new Persona("Alessandro"), 8);  
  
        assertEquals("Alessandro", Collections.min(l).getNome());  
        assertEquals("Valter", Collections.max(l).getNome());  
    }  
}
```

NOTA: se **Persona** non implementasse **Comparable<Persona>**, si sollevarebbe un errore a tempo di compilazione

# L'interface `java.util.Comparator<T>`

- Se vogliamo ordinare una lista secondo un criterio diverso dall'ordinamento naturale?
- La classe **`Collections`** offre una versione del metodo *`sort()`* che si affida ad un **oggetto esterno**, passato come parametro, che sa effettuare i confronti necessari all'ordinamento

```
Collections.sort(  
    List<T> listaDaOrdinare,  
    Comparator<? super T> comparatore  
)
```

# L'interface `java.util.Comparator<T>`

- L'interfaccia `java.util.Comparator<T>` ha un metodo:

```
public int compare(T o1, T o2)
```

che deve restituire un valore che è

- minore, uguale, maggiore di zero a seconda che l'oggetto riferito da `o1` sia
- minore, uguale, maggiore dell'oggetto riferito dal parametro `o2`
- N.B. è simile ma non identico al metodo `compareTo()` di `Comparable<T>`

# Ordinare una lista con `java.util.Comparator<T>`

- Ordinamenti e ricerche possono essere effettuate anche facendo affidamento ad oggetti istanza di classi che implementano **`Comparator<T>`**, un'interface parametrica che prevede il metodo (vedi dettagli nella documentazione):
- `int compare(T o1, T o2)`  
*Compares its two arguments for order.*  
*Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.*

# Esercizio (cont.)

- Supponiamo di voler ordinare una lista di oggetti Persona per età
- Introduciamo (e usiamo) un opportuno comparatore esterno

```
import java.util.Comparator;
```

```
public class ComparatorePersonePerEta
    implements Comparator<Persona> {

    public int compare(Persona p1, Persona p2) {
        return p1.getEta() - p2.getEta();
    }
}
```



# Esercizio (cont.)

```
import static org.junit.Assert.*;

import org.junit.Test;

public class ComparaPersonePerEtaTest {

    @Test
    public void testCompare() {
        Persona paolo = new Persona("Paolo", 61);
        Persona anna = new Persona("Anna", 55);
        ComparatorePerEta comparator =
            new ComparatorePersonePerEta<Persona> ();

        assertTrue(comparator.compare(paolo, anna) > 0);
        assertTrue(comparator.compare(anna, paolo) < 0);
        assertEquals(0, comparator.compare(paolo, paolo));
        assertEquals(0, comparator.compare(anna, anna));
    }
}
```

# Ordinare una lista secondo un criterio diverso dall'ordinamento naturale

```
public class SortTest {  
    @Test  
    public void testSort() {  
        List<Persona> l = new LinkedList<Persona>();  
        l.add(new Persona("Valter", 5));  
        l.add(new Persona("Paolo", 10));  
        l.add(new Persona("Giacomo", 7));  
        l.add(new Persona("Alessandro", 8));  
        ComparatorePersonePerEta<Persona> comparatore =  
            new ComparatorePersonePerEta<Persona>();  
        Collections.sort(l, comparatore);  
        assertEquals("Valter", l.get(0).getNome());  
        assertEquals("Giacomo", l.get(1).getNome());  
        assertEquals("Alessandro", l.get(2).getNome());  
        assertEquals("Paolo", l.get(3).getNome());  
    }  
}
```

# Ordinamenti

- In sostanza, se abbiamo bisogno di operare ordinamenti (o altre operazioni basate su una relazione d'ordine) su una lista **List<T>**
  - Gli elementi della lista devono implementare l'interface **java.lang.Comparable<T>**: la relazione d'ordine rappresentata da questa implementazione corrisponde all'ordinamento naturale degli elementi
- Se abbiamo bisogno di effettuare ordinamenti su relazioni d'ordine diverse da quella naturale, allora possiamo definire una implementazione di **java.util.Comparator<T>**

# Nota

- L'interface `Comparable<T>` è nel package `java.lang`, quindi non è necessario importarla
- L'interface `java.util.Comparator<T>` è nel package `java.util`, quindi va importata