

Programmazione Orientata agli Oggetti

Collezioni
Liste

Sommario

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

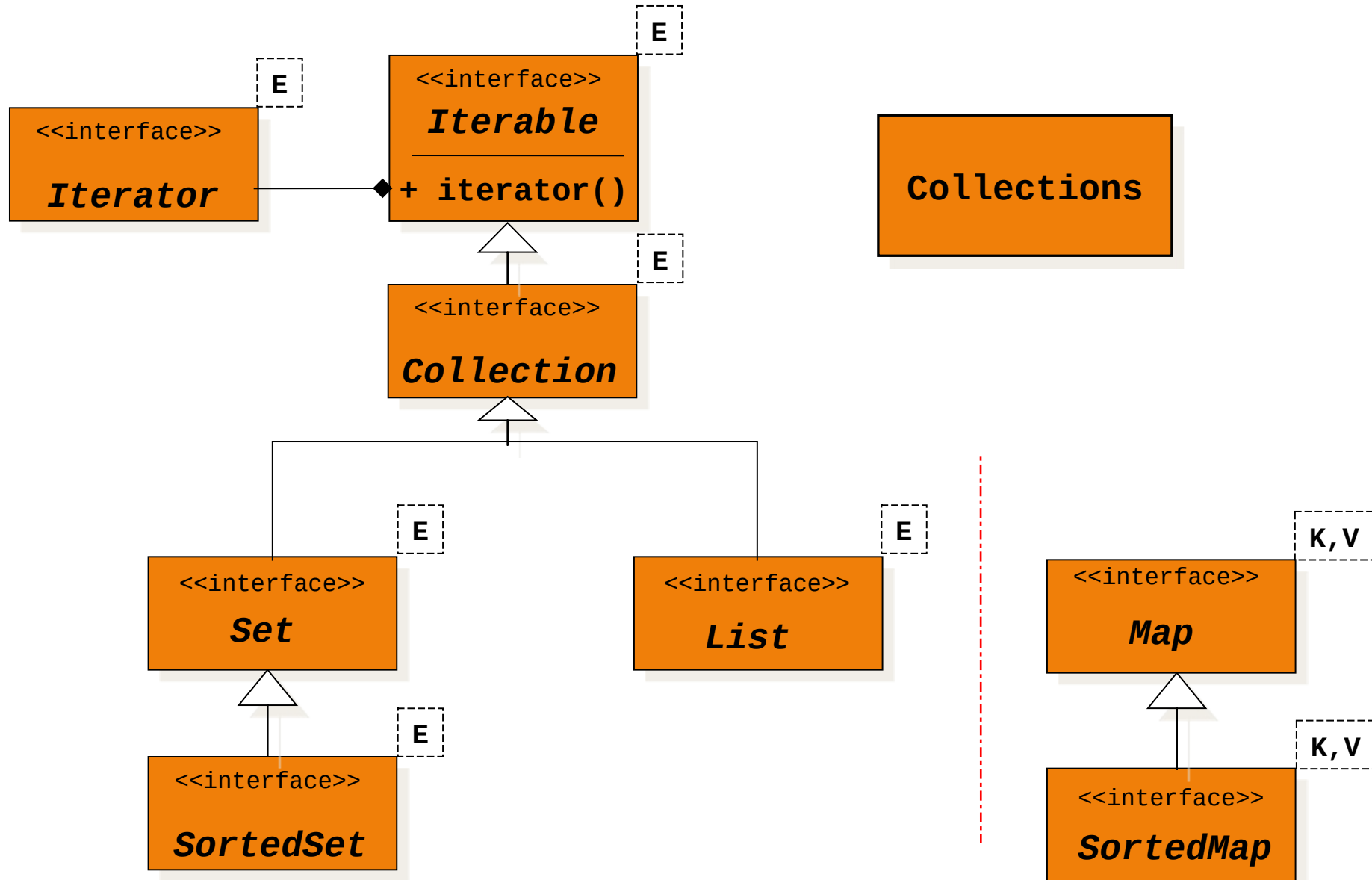
Introduzione

- 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 finora abbiamo ipotizzato un numero massimo di elementi proprio per facilitarne l'utilizzo
 - un indicatore di riempimento serve a ricordarsi il numero di elementi già 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
- Introdotte già in Java 2:
 - ma sostanzialmente rivisitate in seguito alla introduzione dei Generics in Java 5
 - ✓ N.B. I Generics di Java 5 sono stati introdotti principalmente per il loro utilizzo nelle collezioni preesistenti
 - significativamente *estese* nelle versioni successive
 - ✓ per coprire scenari di utilizzo via via conclamatesi come importanti
 - ad es. `java.util.concurrent`: collezioni concorrenti
- ✓ Certamente tra le librerie più utilizzate in assoluto

Il Java Collection Framework (JCF)



Un Primo Sguardo: Collection<E> (1)

- L'interface **Collection<E>** dichiara i metodi di una generica collezione:
- Generalizza sia **List<E>** sia **Set<E>**

List<E>:

- Collezioni sequenziali i cui elementi possiedono una posizione
- Senza gestione dei duplicati

Set<E>:

- Collezioni che non ammettono duplicati
- Gli elementi non possiedono posizione

Un Primo Sguardo: Collection<E> (2)

- 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

Un 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*
 - sarà necessario utilizzare un meccanismo di modellazione del criterio di equivalenza tra elementi dell'insieme
 - bisognerà utilizzarlo per definire un criterio di equivalenza tra gli elementi ospitati nell'insieme

Un Primo Sguardo: `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<E>`**, l'interface **`List<E>`** include altre operazioni specifiche, quali:
 - *Accesso posizionale*: permette di accedere agli elementi di tipo **`E`** in base alla loro posizione nella lista (in maniera simile a quanto avviene per gli array)
 - *Ricerca*: permette di ricercare la posizione di un elemento nella lista

Un Primo Sguardo: 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:
 - *Accesso per chiave*: 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

Un Primo Sguardo: Implementazioni

- **List<E>**
 - **ArrayList<E>**
 - **LinkedList<E>**
- **Set<E>**
 - **HashSet<E>**
 - **TreeSet<E>**
- **Map<K, V>**
 - **HashMap<K, V>**
 - **TreeMap<K, V>**

... le più diffusamente utilizzate, ma ne esistono molte altre di uso più specifico

Un Primo Sguardo: Collections

- La classe **java.util.Collections** (al plurale: attenzione alla 's' finale!)
 - offre un vasto insieme di metodi (statici) generici che implementano utili e diffusi algoritmi per la manipolazione di liste quali:
 - ordinamento
 - ricerca max e min
 - shuffle (mescolamento casuale)
 - reverse
 - fill...
- ✓ A meno di **forti** (anzi fortissime) motivazioni in senso contrario, implementare funzionalità equivalenti a questi (o ad uno degli altri) metodi statici offerti da **Collections** è solo una perdita di tempo

Sommario


- Introduzione alle Collezioni
 - **Interface Collection<E>**
 - Iterare una collezione: **Iterator<E>**
 - Rimuovere elementi da una collezione
- Liste
 - 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 di tre categorie:
 - Manipolazione di base
 - Bulk
 - Conversione da e verso array

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);  
}
```



A dashed box containing a red question mark is positioned to the right of the `contains` method signature. Dotted lines with arrows point from this box to the `contains` method signature and the `containsAll` method signature.

Collection<E>: Metodi Base

Consultare i Javadoc! In sintesi:

- **boolean isEmpty();**
ritorna **true** se la collezione è vuota
- **int size();**
ritorna il numero di elementi presenti nella collezione
- **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

Consultare i Javadoc! In sintesi:

- **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

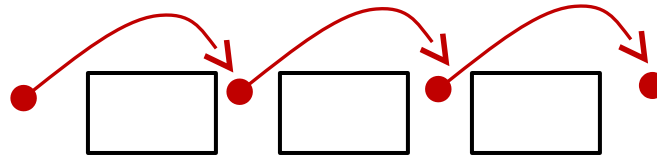
- Introduzione alle Collezioni
 - Interface **Collection<E>**
 - Iterare una collezione: **Iterator<E>**
 - Rimuovere elementi da una collezione
- Liste
 - aggiungere elementi
 - iterare sugli elementi della lista
- Ordinamento di liste
 - **Comparable, Comparator**

Iterazione: Interface `Iterator<E>`

- L'iterazione di una collezione avviene attraverso un oggetto *iteratore* dedicato allo scopo
- Gli iteratori sono creati invocando il factory method **`iterator()`** sulla collezione che si vuole scandire
- L'oggetto ottenuto implementa l'interface **`Iterator<E>`**, munita dei metodi
 - **`boolean hasNext()`**
 - **`E next()`**
 - **`void remove()`**>>
- ✓ Solo i primi due metodi sono considerati strettamente caratterizzanti gli iteratori

Semantica degli Iteratori

- Nella sostanza, un *cursore* che scandisce la collezione sottostante ricordando la sua posizione nella scansione
- Posizioni lecite: subito *prima* o subito *dopo* un elemento della collezione che si sta iterando
 - ✓ N.B. mai “sopra” un elemento



Collezioni di Tante Tipologie, un Unico Modo per Enumerarle

- ✓ L'iterazione si effettua sempre nello stesso identico modo indipendentemente dal tipo di collezione sottostante che lo ha *generato*
- Con notevole “economia di pensiero”
- Per questo motivo è possibile discuterli ancora prima delle implementazioni che sanno generarli, con riferimento all'interface **Collection<E>**

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

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

Esercizio: la Semantica di Iterator

- Per comprendere la semantica dei metodi di una classe non esiste metodo migliore di una batteria di test-case che la documenti precisamente
- Per **Iterator<E>** scriviamo test per i due metodi principali dell'interfaccia **Iterator<E>**
- Utilizziamo come implementazione concreta della collezione **ArrayList<E>** (**>>**)

Unit-Testing per Documentare la Semantica di Iterator<E>

```
import ...  
public class IteratorTest {  
  
    private List<String> vuota;  
    private List<String> singoletto;  
    private String solitario;  
  
    @Before  
    public void setUp() {  
        this.vuota = new ArrayList<>();  
        this.singoletto = new ArrayList<String>();  
        this.solitario = new String("solitario");  
        this.singoletto.add(this.solitario);  
    }  
  
    @Test ... ..  
}
```

// da Java 7
ArrayList<>();

Diamond Operator

Test di `Iterator.hasNext()`

`@Test`

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

`@Test`

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

Test di `Iterator.next()`

`@Test`

```
public void testNext_singoletto() {  
    Iterator<String> it =  
        this.singoletto.iterator();  
    assertNotNull(it);  
    assertTrue(it.hasNext());  
    assertSame(this.solitario, it.next());  
}
```

Test di Iterator.next()

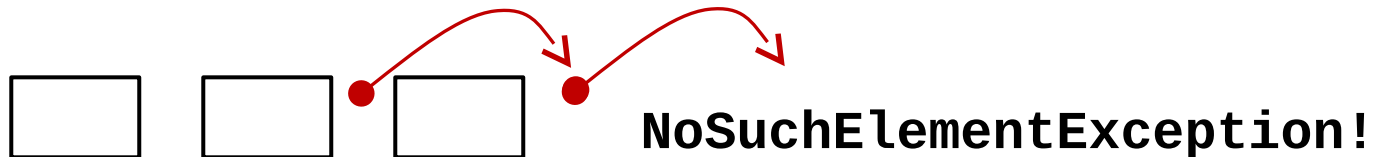
@Test

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

Test di `Iterator.next()`

```
@Test(expected = NoSuchElementException.class)
public void testNext_oltreLaFineSollevaEccezione() {
    Iterator<String> it = this.vuota.iterator();
    it.next();
}
```

- Posizioni lecite: subito *prima* o subito *dopo* un elemento della collezione che si sta iterando
 - ✓ raggiunto l'ultimo elemento non si può andare oltre: il metodo **`Iterator.next()`** solleva una eccezione (>>)



Iterazione di Array - Ciclo *for*

```
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 = this.attrezzi[i];  
            pesoTotale += a.getPeso();  
        }  
        return pesoTotale;  
    }  
    ...  
}
```

Iterazione Mediante Iteratori

```
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 = iteratore.next();
            pesoTotale += a.getPeso();
        }
        return pesoTotale;
    }
    ...
}
```

Esercizio List<E>: Iterazione

- L'utilizzo degli iteratori è decisamente ripetitivo
- Sempre le stesse identiche operazioni:
 - **Iterator<Attrezzo> iteratore =**
this.attrezzi.iterator();
Abbiamo chiesto alla lista di creare un oggetto per gestire l'iterazione su una collezione di oggetti **Attrezzo...**
 - **while (iteratore.hasNext())**
...fintanto che ci sono ancora elementi da scandire...
 - **Attrezzo a = iteratore.next();**
...attraverso il metodo **next()** otteniamo dall'iteratore il prossimo elemento della scansione
- E' ben motivata una forma sintattica abbreviata...

Iterazione: *for-each* (1)

- Per iterare su **tutti** gli elementi di una collezione è possibile usare la forma "for-each" dell'istruzione **for**

```
for( Tipo elemento : iterable )  
    <<blocco_di_operazioni_su_elemento>>
```
- Dove *iterable* è un qualsiasi sottotipo di **java.lang.Iterable<E>**, una interface che offre il factory method:
 - **Iterator<E> iterator()**
- ✓ La sintassi for-each è conveniente *solo se non* è necessario accedere all'indice di iterazione

Iterazione: *for-each* (2)

- ✓ Si tratta solo di “zucchero sintattico”: il compilatore traduce

```
for( E elemento : iterable )
    <<blocco_di_codice_su elemento>>
... in ...
for (Iterator<E> iter = iterable.iterator();
    iter.hasNext(); ) {
    E elemento = iter.next();
    <<blocco_di_codice_su elemento>>
}
```
- Tutte le collezioni del JCF sono già *Iterable*
 - Si può usare il *for-each* su qualunque classe, anche di nuova definizione, purché implementi **Iterable<E>**
 - Ed anche sugli array che pure non lo sono affatto (sottotipi di **Iterable<E>**)
 - ✓ grazie ad una gestione peculiare e specifica da parte del compilatore

Esercizio:

Scansione di Liste con *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;
    }
    ...
}
```

Sommario

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

Legame Iterator / Collezione

- Tra un iteratore e la collezione che lo ha creato permane un legame anche successivamente alla sua creazione
 - Interessante anche per i meccanismi di costruzione dei tipi utilizzati nell'occasione (>>)
- L'esistenza di un terzo metodo nell'interface **Iterator<E>** per la rimozione di elementi rende più evidente la natura di questo legame:

void remove();

- rimuove dalla collezione l'ultimo elemento che è stato restituito da una precedente chiamata di **next()**
- ✓ attraverso l'iteratore vengono operate modifiche sulla collezione sottostante

Test di `Iterator.remove()`

@Test

```
public void testRemove() {  
    Iterator<String> it =  
        this.singoletto.iterator();  
    String solitario = it.next();  
    assertFalse(this.singoletto.isEmpty());  
    it.remove();  
    assertTrue(this.singoletto.isEmpty());  
}
```

Il Metodo `remove()` di Iterator

- Il metodo **`remove()`** rimuove l'elemento restituito dall'ultima chiamata di **`next()`**
- Non è ammesso chiamare **`remove()`** a meno che prima non si sia provveduto ad invocare **`next()`**
- Ad es. per eliminare due elementi consecutivi:

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

prima bisogna richiamare **`next()`**:

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

Rimuovere Elementi da una Collezione

- Pertanto esistono due diversi modi per rimuovere un elemento da una collezione, ciascuno dettato da specifiche esigenze:
 - per la rimozione di un elemento uguale (secondo quanto stabilito dal metodo **boolean equals(Object o)**) 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 si usa il metodo **remove()** di **Iterator**

Il Metodo `remove(Object o)` di `Collection<E>`

`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<E>`

- **`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)

Come Rimuovere Elementi?

- Attenzione: è un errore cercare di rimuovere elementi da una collezione con il metodo
 - **boolean remove(Object o)**di **Collection** proprio mentre si sta ancora visitando la stessa collezione con un iteratore
 - la collezione verrebbe modificata "sotto i piedi" dell'iteratore che la sta ancora scandendo
- Se si stanno cercando elementi da rimuovere attraverso un iteratore, *deve* poi essere usato il metodo **remove()** dell'iteratore stesso per renderlo «consapevole» dei cambiamenti alla «sua» collezione

java.util.ConcurrentModificationException

```
import java.util.*;
public class ConcurrentModificationMain {
    public static void main(String args[]) {
        List<Object> list = new ArrayList<>();
        Iterator it = list.iterator();
        list.add(new Object());
        it.next(); // Qui solleva ConcurrentModificationException
    }
}
```

- ✓ L'eccezione è sollevata solo alla prima occasione utile (semantica *fail-fast*, vedere Javadoc)

Exception in thread "main"

java.util.ConcurrentModificationException

at java.util.AbstractList\$Itr.checkForComodification(AbstractList.java:448)

at java.util.AbstractList\$Itr.next(AbstractList.java:419)

at ConcurrentModificationMain.main(ConcurrentModificationMain.java:9)

Sommario

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

Liste: Interface List<E> (1)

- 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)
- Cfr. ASD (<<)

Liste: Interface List<E> (2)

- L'interface **List<E>** estende l'interface **Collection<E>**
- Oltre ai metodi della interface **Collection<E>**, **List<E>** offre specifici 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*

Esercizio (per Casa)

- Analizzare, compilare ed eseguire la classe di test **ListTest** riportata subito di seguito.
 - Aggiungere opportuni metodi di test per verificare e comprendere la semantica dei metodi:
 - **indexOf(Object o);** in particolare cerca la stessa istanza in memoria od un oggetto **equals()** ???
 - **contains(Object o);** idem...
 - **retainAll(Collection<?> c)**
 - **containsAll(Collection<?> c)**

Esercizio (per Casa, *continua*)

```
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());
    }
}
```

Implementazioni di List<E>

- Il package `java.util` offre due diverse implementazioni di `List<E>`
 - `ArrayList<E>`
 - `LinkedList<E>`
- Forniamo qualche (grossolana) indicazione su come scegliere l'implementazione più opportuna
- ✓ **NOTA:** Questi aspetti sono stati già approfonditi nel corso "*Algoritmi e Strutture Dati*"

ArrayList<E>: Implementazione

- Gli elementi sono memorizzati in un contenitore rappresentato con array ed indicatore di riempimento
 - Al momento della creazione, la dimensione dell'array (ovvero la capacità della collezione) è inizializzata ad un valore prestabilito
 - 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 travasati tutti gli elementi dell'array originario
- ✓ 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
- Nella pratica la complessità delle JVM moderne rende le differenze spesso impercettibili e/o comunque molto difficilmente prevedibili
 - ✓ Basare le ottimizzazioni *sempre* su misurazioni sperimentali che ne palesino ed accertino la reale necessità

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<E>()** *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<E>(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*

Esercizio: List<E>

- La classe **ArrayList<E>** implementa l'interfaccia **List<E>** (e quindi è sottotipo anche di **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

Esercizio List<E>: Aggiunta di Elementi

```
public class Borsa {  
    private Attrezzo[] attrezzi;  
    private int numeroAttrezzi;  
    public Borsa() {  
        this.numeroAttrezzi = 0;  
        this.attrezzi = new Attrezzo[10];  
    }  
    public boolean addAttrezzo(Attrezzo attrezzo){  
        if (numeroAttrezzi == 10)  
            return false;  
        this.attrezzi[this.numeroAttrezzi] = attrezzo;  
        this.numeroAttrezzi++;  
        return true;  
    } ...  
}
```

array

```
import java.util.List;  
import java.util.ArrayList;  
public class Borsa {
```

```
    private List<Attrezzo> attrezzi;  
    public Borsa() {  
        this.attrezzi = new ArrayList<Attrezzo>();
```

ArrayList

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

```
}
```


Esercizio List<E>: Osservazioni

- Dobbiamo importare:

`java.util.List`

`java.util.ArrayList`

- Principali benefici rispetto all'uso degli array
 - 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

Esercizio List<E>: Aggiunta di 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

Esercizio List<E>: Osservazioni

- La lista aumenta la sua capacità se necessario
- Mantiene un conteggio del numero di elementi
- Il metodo **int size()** restituisce questo valore
- ✓ I dettagli di come tutto ciò viene realizzato ci viene nascosto
 - È importante?
 - Non conoscere questi dettagli ci impedisce di usare la collezione?
- ✓ Quanto risulta difficile cambiare la scelta dell'implementazione?
 - Ad es. passare da **ArrayList** a **LinkedList**...

Metodi Specifici di `LinkedList<E>`

- `LinkedList<E>` offre anche alcuni metodi “fuori” dalla interface `List<E>` e quindi non offerti anche da `ArrayList<E>`
 - Similarmente `ArrayList<E>` offre costruttori (basati sulla *capacità*) che `LinkedList<E>` non offre
- Sono metodi tesi ad evidenziare l’accesso efficiente da parte delle `LinkedList<E>` in testa ed in coda
- Ad es.
 - `addFirst()/Last()`
 - `getFirst()/Last()`
 - `removeFirst()/Last()`
 - ...
- ✓ Attenzione: utilizzarli rende meno intercambiabili le due implementazioni
- Per questo si consiglia di dichiarare i tipi, a meno di *forti* motivazioni in senso contrario, utilizzando *sempre* le interface

Esercizio List<E>: Rimozione di un Elemento

```
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: `ListIterator<E>`

- In realtà l'interface ***List*** affianca al factory method **`iterator()`** di ***Collection*** un metodo specifico per le liste (e non esistente per tutte le generiche collezioni)
- Il metodo **`List<E>.listIterator()`** restituisce un ***ListIterator<E>***
 - ✓ ***ListIterator*** estende ***Iterator***
- Studiare (consultando i javadoc) le differenze tra le due interface ***Iterator*** e ***ListIterator***
- Scrivere dei test di unità sulla semantica dei metodi di ***ListIterator*** verificando che continuino ad aver successo anche i test-case già scritti per il suo supertipo ***Iterator***

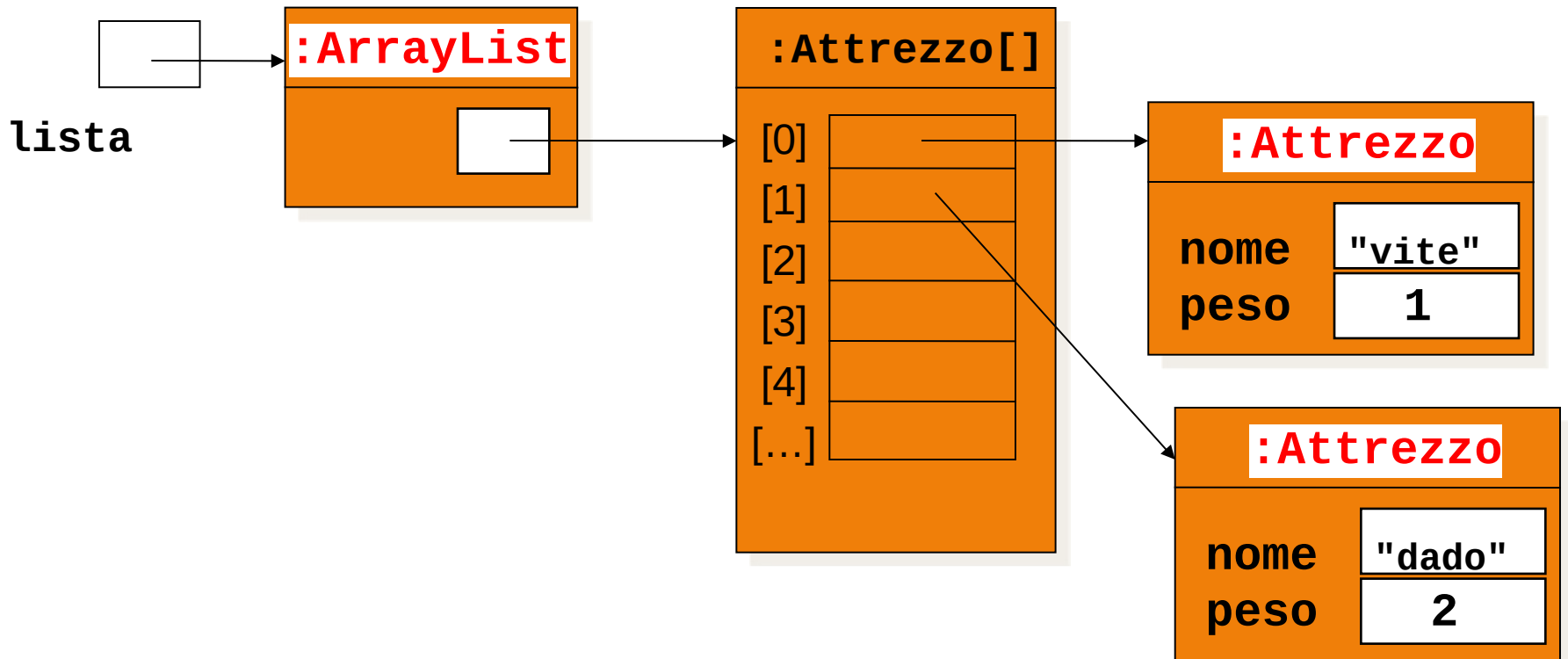
Liste: Diagramma degli Oggetti

- 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

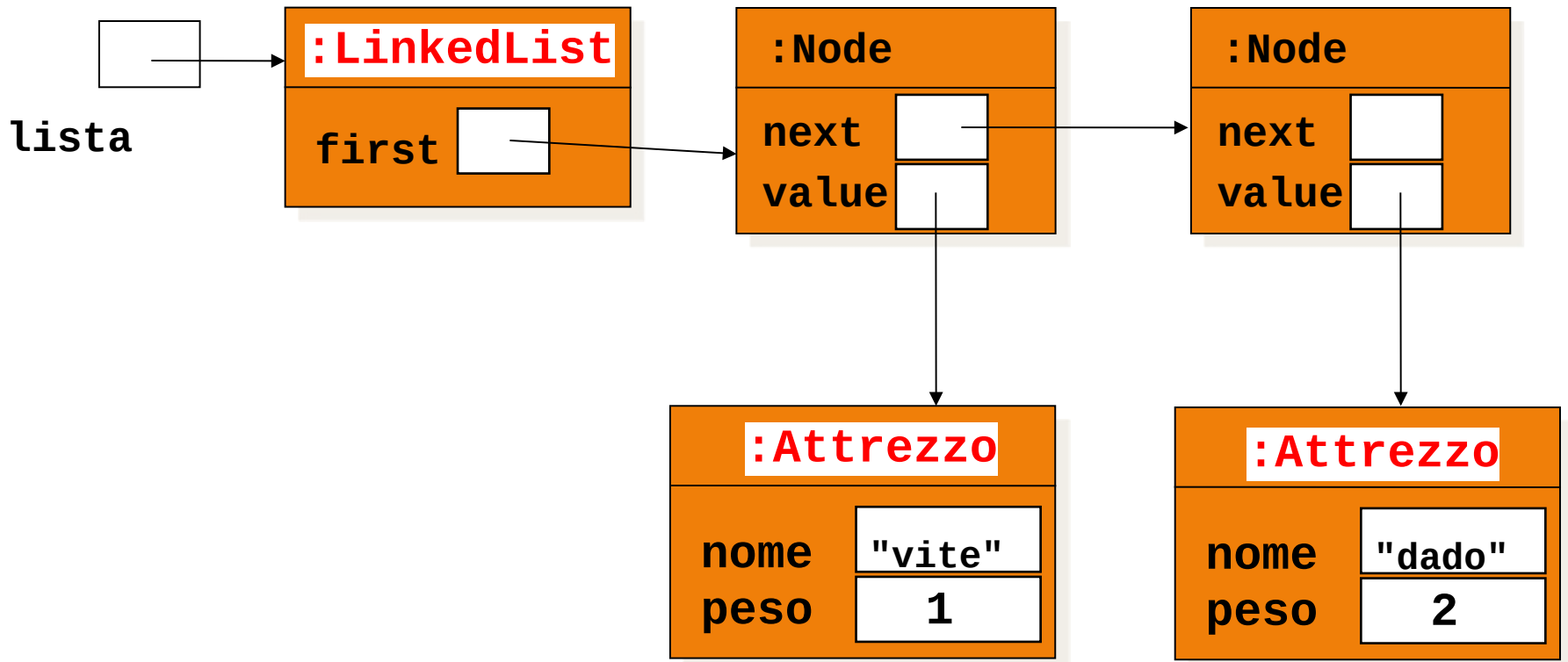
```
List<Attrezzo> lista;  
lista = new ArrayList<Attrezzo>();  
lista.add(new Attrezzo("vite",1);  
lista.add(new Attrezzo("dado",2);
```

// DIAGRAMMA DA DISEGNARE QUANDO L'ESECUZIONE RAGGIUNGE QUESTO PUNTO



LinkedList: Diagramma degli Oggetti

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



Sommario

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

Ordinamenti e Ricerche

- Il JCF (in particolare la classe **Collections**) include metodi che implementano algoritmi efficienti per
 - ordinare una lista
 - ricercare la posizione di un elemento in una lista ordinata
 - ricercare il min/max in una lista

Relazione d'Ordine

- Queste operazioni sono possibili solo se esiste una relazione d'ordine per il tipo degli elementi ospitati nella collezione
 - 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
- ✓ Anche una relazione d'ordine su un supertipo degli elementi ospitati va bene

Definizione di un Criterio di Ordinamento

- 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<T>`**
(il cosiddetto ordinamento «*naturale*» o «*interno*»)
 - 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<T>`**
(il cosiddetto ordinamento «*esterno*»)

L'Interface `java.lang.Comparable<T>`

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

`public int compareTo(T that)`

- Restituisce un valore che è:
 - minore, uguale, maggiore di zero a seconda che l'oggetto corrente **`this`** sia rispettivamente:
 - minore, uguale, maggiore dell'oggetto il cui riferimento è ricevuto come parametro formale **`that`**

Utilizzi di `java.lang.Comparable<T>`

- Molte importanti classi della libreria standard già implementano l'interfaccia **`java.lang.Comparable<T>`**, adottando una semantica più o meno scontata.
- Ad 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 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;  
    }  
  
    @Override  
    public int compareTo(Persona that) {  
        return this.nome.compareTo(that.getNome());  
    }  
}
```


Test di Persona.compareTo()

```
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);  
        assertEquals(0, p1.compareTo(p3)); // 0  
  
        Persona p4 = new Persona("Anna", 8);  
        assertTrue(p1.compareTo(p4) > 0);          // >0  
    }  
  
}
```

Ordinamento «Naturale»

- Un ordinamento naturale di oggetti è definito dalla relazione d'ordine implementata dal metodo **compareTo()** nell'interface **Comparable<T>**
- E' possibile quindi operare su una **List<T>** contenente oggetti che implementano **Comparable<T>** con metodi che utilizzino tale criterio di ordinamento:
 - si possono effettuare ricerche efficienti
 - si può calcolare il massimo e il minimo
 - si può effettuare l'ordinamentomediante i metodi offerti da **Collections**

Ordinamento di Liste: `Collections.sort()`

- Metodo statico **`Collections.sort()`**, in due versioni sovraccariche corrispondenti ai due diversi modi («naturale» vs «esterno») di fornire un criterio di ordinamento
- Segnatura per l'ordinamento naturale:

```
public static <T extends Comparable<? super T>>  
    void sort(List<T> list)
```

Test dell'Ordinamento «Naturale»

```
public class OrdinamentoNaturaleTest {  
    @Test  
    public void testSort() {  
        List<Persona> l = new LinkedList<>();  
        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());  
    }  
}
```

Se **Persona** non implementasse **Comparable<Persona>**,
si solleverebbe un errore già a tempo di compilazione

Ottenere *Min* & *Max* di una Lista

- Data una lista **List<T>** i cui elementi implementino l'interface **Comparable<T>**, può essere calcolato l'elemento massimo/minimo (rispetto all'ordinamento naturale) mediante i metodi statici di **Collections**:
 - **Collections.min()**
 - **Collections.max()**

```
public static <T extends Object & Comparable<? super T>>  
    T min/max(Collection<? extends T> coll)
```

...che semplifichiamo in:

```
public static <T extends Comparable<? super T>>  
    T min/max(Collection<? extends T> coll)
```

Test del Calcolo Min & Max di una Lista

```
public class MinMaxTest {  
    @Test  
    public void testMinMax() {  
        List<Persona> l = new LinkedList<>();  
  
        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());  
    }...  
}
```

✓ Se **Persona** non implementasse **Comparable<Persona>**,
si solleverebbe un errore a tempo di compilazione

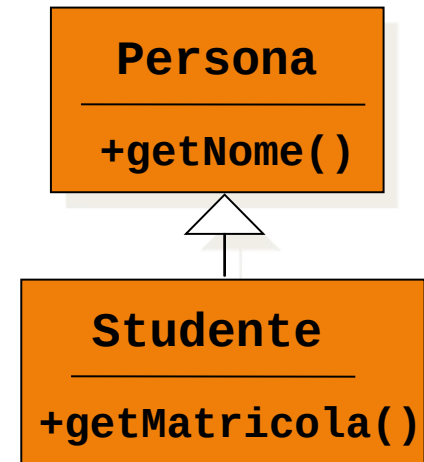
Ordinamento «Naturale»: Limiti (1)

- L'ordinamento naturale può essere definito al più una sola volta in ogni classe
- In effetti può essere utilizzata una sola volta per ogni intera *gerarchia di tipi* !
- Per capire perché, consideriamo due classi, **Persona** e **Studiante**, ove **Studiante** estende **Persona**, e prevediamo per entrambe un ordinamento «naturale»
 - le persone sono ordinate per nome
 - gli studenti sono ordinati per matricola

Ordinamento «Naturale»: Limiti (2)

```
public class Persona implements Comparable<Persona> {  
    private String nome;  
    public Persona(String nome) { this.nome = nome; }  
    public String getNome() { return this.nome; }  
    @Override  
    public int compareTo(Persona that) {  
        return this.getNome().compareTo(that.getNome());  
    }  
}
```

```
public class Studente extends Persona implements Comparable<Studente> {  
    private String matricola;  
    public Studente(String nome, String matricola) {  
        super(nome);  
        this.matricola = matricola;  
    }  
    public String getMatricola() { return this.matricola;  
    @Override  
    public int compareTo(Studente that) {  
        return this.getMatricola().compareTo(that.getMatricola());  
    }  
}
```



NON COMPILA!

The interface **Comparable** cannot be implemented more than once with different arguments: **Comparable<Persona>** and **Comparable<Studente>**

Limitazioni dei Java Generics (1)

- Alla stessa classe/interface (in generale tipo) non è permesso implementare la stessa interface generica più di una volta con tipi attuali distinti!
- Chiaramente controintuitivo; contraddice almeno questi aspetti che lasciavano sperare altrimenti:
 - una classe Java può implementare diverse interface
 - si possono definire metodi sovraccarichi di stesso nome ma diversa segnatura per l'uso di tipi polimorfi (anche se pescati dalla medesima gerarchia)
- In effetti è come se la piattaforma si rifiutasse di considerare diversi due tipi generici validi se differiscono *solo* per uno dei tipi attuali utilizzati
- ✓ Ed è esattamente così...

Limitazioni dei Java Generics (2)

- *java.lang.Comparable* presente sin da Java 1.0
 - ✓ ovviamente non utilizzava i generics: introdotti solo in Java 5
 - ✓ Si basava su **Object**: ***Comparable*.boolean compareTo(Object o)**
- L'*irrinunciabile* retrocompatibilità ha imposto forti assunzioni sull'uso dei generics: in particolare NON risultò possibile cambiare le informazioni sui tipi (dinamici) disponibili a tempo di esecuzione rispetto alle versioni pre-generics
 - Detto diversamente, ciascun tipo deve essere *sempre* distinguibile a tempo *dinamico* anche se cancelliamo i tipi attuali utilizzati nella definizione dei tipi generici e finalizzati a tempo *statico*
 - A tempo dinamico non rimane traccia dei generics, che sono sempre risolti in un tipo non generico (chiamato «erasure») già a tempo di compilazione
- ✓ Per motivazioni similari, non è possibile creare array generics
 - Per es. **T[] ag = new T[10]; // T tipo NON COMPILA**

Limitazioni dell'Ordinamento «Naturale» o «Interno»

- A causa di questa sfortunata interazione con le limitazioni dei generics, l'ordinamento naturale può essere definito in uno solo dei tipi di una gerarchia
 - nell'es. di prima, dentro **Studente** oppure dentro **Persona**, ma non in entrambe
- Pertanto, considerando:
 - la necessità di definire molteplici criteri di ordinamento su oggetti dello stesso tipo (o della stessa gerarchia di tipi)
 - nonché la necessità di disaccoppiare la definizione del criterio di ordinamento da quella della classe le cui istanze risulteranno ordinate
- ✓ risulta ben motivato l'utilizzo di soluzioni (per la definizione di un criterio di ordinamento) «esterne» alla classe

Ordinamento «Esterno»

- Se vogliamo ordinare una lista secondo un criterio diverso dall'ordinamento naturale?
- Segnatura per l'ordinamento esterno:

```
public static <T> void sort(  
    List<T> listaDaOrdinare,  
    Comparator<? super T> comparatore  
)
```
- Si affida ad un *oggetto esterno*, passato come parametro, per effettuare i confronti necessari all'ordinamento
 - ✓ unica soluzione possibile quando serve più di un criterio di ordinamento

L'Interface

`java.util.Comparator<T>`

- L'interfaccia `java.util.Comparator<T>` consiste di un solo 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**
- ✓ (per ricordarselo: “come fosse **o1** - **o2** per un ord. crescente”)
- ✓ N.B. la segnatura è simile ma *non* identica a quella del metodo **compareTo()** di **Comparable<T>**

Esercizio (cont.)

- Supponiamo di voler ordinare una lista di oggetti **Persona** per età
- Introduciamo un opportuno comparatore esterno:

```
import java.util.Comparator;

public class ComparatorePerEta
    implements Comparator<Persona> {

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

}
```

Test di Comparator<Persona>.compare()

```
// import omessi
```

```
public class ComparatorePerEtaTest {
```

```
    @Test
```

```
    public void testCompare() {
```

```
        Persona paolo = new Persona("Paolo", 61);
```

```
        Persona anna = new Persona("Anna", 55);
```

```
        ComparatorePerEta comparator =  
            new ComparatorePerEta();
```

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

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

```
        assertEquals(0, comparator.compare(paolo, paolo));
```

```
        assertEquals(0, comparator.compare(anna, anna));
```

```
    }
```

```
}
```

Test dell'Ordinamento «Esterno»

```
public class OrdinamentoEsternoTest {
    @Test
    public void testSort() {
        List<Persona> l = new LinkedList<>();

        l.add(new Persona("Valter", 5));
        l.add(new Persona("Paolo", 10));
        l.add(new Persona("Giacomo", 7));
        l.add(new Persona("Alessandro", 8));
        ComparatorePerEta comparatore =
            new ComparatorePerEta();
        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());
    }
}
```


Note Finali

- 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 esplicitamente
- In **Collections** esistono anche i metodi per il calcolo del min/max secondo l'ordinamento esterno.

Ad es.:

```
static <T> T min(Collection<? extends T> coll,  
                Comparator<? super T> comp)
```

Esercizio

- Senza cambiare la classe **Libro** (riportata di seguito), scrivere il metodo **elencoOrdinatoPerPagine()** della classe **Biblioteca** affinché restituisca l'elenco dei libri ordinato per numero di pagine

Esercizio (cont.)

```
public class Libro implements Comparable<Libro> {  
  
    private String titolo;  
    private int pagine;  
  
    public Libro(String titolo, int pagine) {  
        this.titolo = titolo;  
        this.pagine = pagine;  
    }  
  
    public String getTitolo() {  
        return this.titolo;  
    }  
  
    public int getPagine() {  
        return this.pagine;  
    }  
    @Override  
    public int compareTo(Libro libro) {  
        return this.getTitolo().compareTo(libro.getTitolo());  
    }  
}
```

Esercizio (cont.)

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

public class Biblioteca {

    private List<Libro> elenco;

    public Biblioteca() {
        this.elenco = new ArrayList<>();
    }

    public void aggiungiLibro(Libro libro) {
        this.elenco.add(libro);
    }

    public List<Libro> elencoOrdinatoPerPagina() {
        ComparatorePerPagina comp = new ComparatorePerPagina();
        Collections.sort(this.elenco, comp);
        return this.elenco;
    }
}
```

Esercizio (cont.)

```
public class ComparatorePerPagine
    implements Comparator<Libro> {

    @Override
    public int compare(Libro l1, Libro l2) {
        return l1.getPagine() - l2.getPagine();
    }

}
```

Conclusioni

- Introduzione alle Collezioni
 - Interacce principali
 - Principali implementazioni
 - Iteratori, e **for-each**
- Java Collection Framework
- Liste
- Ordinamento di liste
 - Ordinamento *Naturale*
 - Ordinamento *Esterno*
 - Generics nelle gerarchie di tipi