Collezioni in Java: il Java Collections Framework (JCF)

Perché le collezioni

- Spesso in un programma dobbiamo rappresentare e manipolare gruppi di valori oppure oggetti di uno stesso tipo
  - insieme di studenti di una classe
  - lista degli ultimi SMS arrivati sul cellulare
  - l'insieme dei risultati di una query al database
  - la coda dei pazienti in attesa di un'operazione
  - …
- Chiamiamo **collezione** un gruppo di oggetti *omogenei* (cioè dello stesso tipo)

Array come collezioni

- Java (come altri linguaggi) fornisce gli array come tipo di dati primitivo “parametrico” per rappresentare collezioni di oggetti
- **Array**: collezione modificabile, lineare, di dimensione non modificabile
- Ma sono utili anche altri tipi di collezioni
  - modificabili / non modificabili
  - con ripetizioni / senza ripetizioni (come gli insiemi)
  - struttura lineare / ad albero
  - elementi ordinati / non ordinati

Il nostro interesse

- Non è solo un interesse pratico (è utile sapere cosa fornisce Java) …
- ma anche un esempio significativo dell'applicazione dei principi di *data abstraction* che abbiamo visto
- Un po' di contesto
  - JCF (Java Collections Framework)
  - C++ Standard Template Library (STL)
  - Smalltalk collections

Ma non bastavano …

- Vector = collezione di elementi omogenei modificabile e estendibile?
- In principio si … **ma è molto meglio avere una varietà ampia di strutture dati con controlli statici per verificare la correttezza delle operazioni**

Java Collections Framework (JCF)

- **JCF** definisce una gerarchia di interfacce e classi che realizzano una ricca varietà di collezioni
- Sfrutta i meccanismi di astrazione
  - per specifica (vedi ad es. la documentazione delle interfacce)
  - per parametrizzazione (uso di tipi generici)

  per realizzare le varie tipologie di astrazione viste
  - astrazione procedurale (definizione di nuove operazioni)
  - astrazione dai dati (definizione di nuovi tipi – ADT)
  - **iterazione astratta <= lo vedremo in dettaglio**
  - gerarchie di tipo (con implements e extends)
- Contiene anche realizzazioni di algoritmi efficienti di utilità generale (ad es. ricerca e ordinamento)

JCF in sintesi

- Una architettura per rappresentare e manipolare collezioni.
  - Gerarchia di ADT
  - Implementazioni
  - Algoritmi polimorfi
- Vantaggi
  - Uso di strutture standard con algoiritmi testati
  - Efficienza implementazioni
  - Interoperabilità
  - Riuso del software

L'interfaccia `Collection<E>`

- Definisce operazioni basiche su collezioni, senza assunzioni su struttura/modificabilità/duplicati…
- Metodi **opzionali**: `add(E e)`, `remove(Object o)`, `addAll(Collection<? extends E>)`, `clear( )`
- …per definire una classe di collezioni *non modificabili*

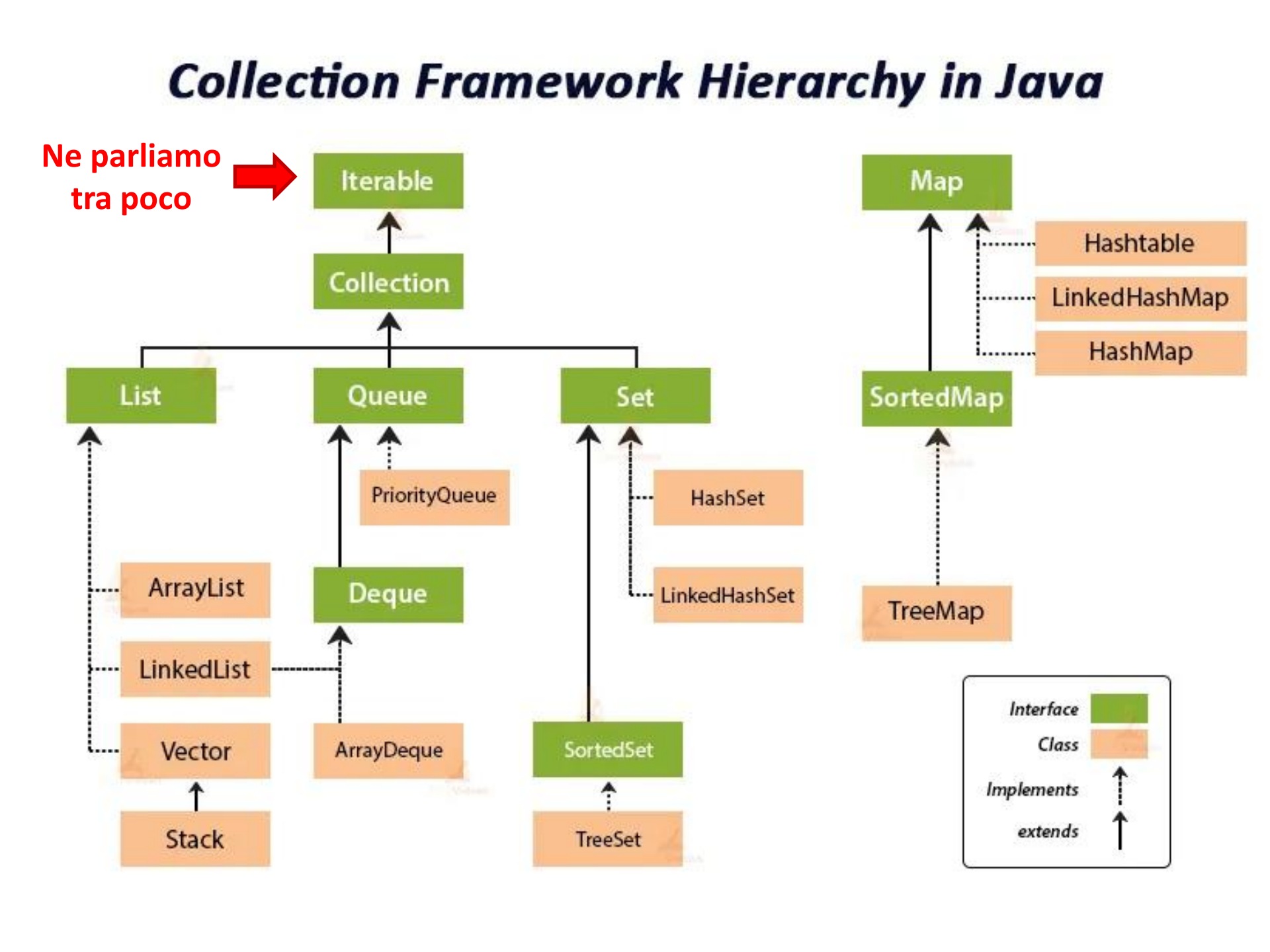
```
public boolean add(E e) {
   throw new UnsupportedOperationException( );
}
```

- Observers: `contains(o)`, `equals(o)`, `isEmpty( )`, `size( )`, `toArray()`
- Accesso agli elementi con `iterator( )` (vedi dopo)

JCF: altre interfacce importanti

- **`Set<E>`**: collezione senza duplicati. Stessi metodi di `Collection<E>`, ma la specifica cambia, ad es.
  - `add(E el)` restituisce false se `el` è già presente
- **`List<E>`**: sequenza lineare di elementi. Aggiunge metodi per operare in una specifica posizione, ad es.
  - `add(int index, E el)`, `int indexOf(el)`, `remove(index)`, `get(index)`, `set(index, el)`
- **`Queue<E>`**: supporta politica FIFO
  - **`Deque<E>`**: "double ended queue". Fornisce operazioni per l'accesso ai due estremi
- **`Map<K,T>`**: definisce un'associazione chiavi (`K`) – valori (`T`). Realizzata da classi che implementano vari tipi di tabelle hash (ad es. `HashMap`)

Collection Framework Hierarchy in Java



JCF: alcune classi concrete

- **`ArrayList<E>, Vector<E>`**: implementazione di `List<E>` basata su array. Sostituisce l'array di supporto con uno più grande quando è pieno
- **LinkedList<E>**: implementazione di `List<E>` basato su *doubly-linked* list. Usa un record type `Node<E>`
  - `Node<E> prev, E item, Node<E> next`
- **`TreeSet<E>`**: implementa `Set<E>` con ordine crescente degli elementi (definito da `compareTo<E>`)
- **`HashSet<E>`**, **`LinkedHashSet<E>`**: implementano `Set<E>` usando tabelle hash

Organizzazione in packages

- Le classi della Java API sono raggruppate in packages
  - **java.lang** riunisce classi fondamentali (es. String, Math,…)
  - **java.util** riunisce classi di frequente utilizzo (es. Random, Scanner (per input da terminale), …)
  - **java.io** riunisce classi per I/O da file
  - **java.net** riunisce classi per applicazioni di rete
  - **java.security** riunisce classi per crittografia e access control
  - **java.awt** e **java.swing** classi per costruire interfacce grafiche
  - … numerosi altri
- Le classi/interfacce del JCF fanno parte del package **java.util**
- Devono essere importate all'inizio del file della classe che le usa

Esempio: `import java.util.HashSet;`

Proprietà di classi concrete

	ArrayList	Vector	LinkedList	HashMap	LinkedHashMap	HashTable	TreeMap	HashSet	LinkedHashSet	TreeSet
Allows Null?	Yes	Yes	Yes	Yes (But One Key & Multiple Values)	Yes (But One Key & Multiple Values)	No	Yes (But Zero Key & Multiple Values)	Yes	Yes	No
Allows Duplicates?	Yes	Yes	Yes	No	No	No	No	No	No	No
Retrieves Sorted Results?	No	No	No	No	No	No	Yes	No	No	Yes
Retrieves Same as Insertion Order?	Yes	Yes	Yes	No	Yes	No	No	No	Yes	No
Synchronized?	No	Yes	No	No	No	Yes	No	No	No	No

JCF: classi di utilità generale

- **java.util.Arrays**: fornisce metodi statici per manipolazione di array, ad es.
  - ricerca binaria e ordinamento: binarySearch e sort
  - operazioni basiche: `copyOf`, `equals`, `toString`
  - conversione in lista [inverso di **`toArray( )`**]:
    `static <T> List<T> asList(T[ ] a)`
  - NB: per far copie di array, usare **`System.arraycopy(…)`**
- **java.util.Collections**: fornisce metodi statici per operare su collezioni, compreso ricerca, ordinamento, massimo, wrapper per sincronizzazione e per immutabilità, ecc.

Iterazione su collezioni: motivi

- Tipiche operazioni su di una collezione richiedono di *esaminare tutti gli elementi, uno alla volta.*
- Esempi: stampa, somma, ricerca di un elemento, minimo …
- Per un array si può usare un **for**

```
for (int i = 0; i < arr.length;i++)
    System.out.println(arr[i]);
```

- Infatti, per array sappiamo
  - la dimensione: quanti elementi contengono (**length**)
  - come accedere in modo diretto a ogni elemento con un indice

Gli iteratori...

- Un iteratore è un'astrazione che permette di estrarre "uno alla volta" gli elementi di una collezione, senza esporne la rappresentazione
- Generalizza la scansione lineare di un array/lista a collezioni generiche
- Sono oggetti di classi che implementano l'interfaccia

```
public interface Iterator<E> {
      boolean hasNext( );
      E next( );
      void remove( );
}
```

…e il loro uso

- Tipico uso di un iteratore (*iterazione astratta*)

```
// creo un iteratore sulla collezione
Iterator<Integer> it = myIntCollection.iterator( );
while (it.hasNext( )) {  // finché ci sono elementi
    int x = it.next( );  // prendo il prossimo
    ...                  // uso  x
}
```

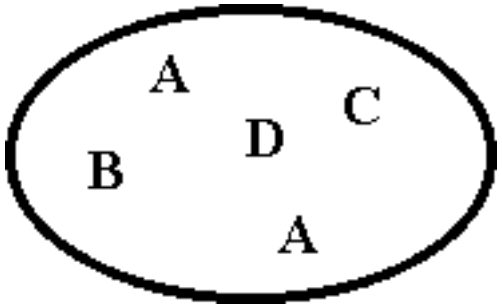
Specifica dei metodi di `Iterator`

```
public interface Iterator<E> {
  boolean hasNext( );
  /* returns: true if the iteration has more elements. (In other words, returns
     true if next would return an element rather than throwing an exception.) */
  E next( );
   /* returns: the next element in the iteration.
       throws: NoSuchElementException - iteration has no more elements. */
  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. */
}
```

Interpretazione grafica - 1

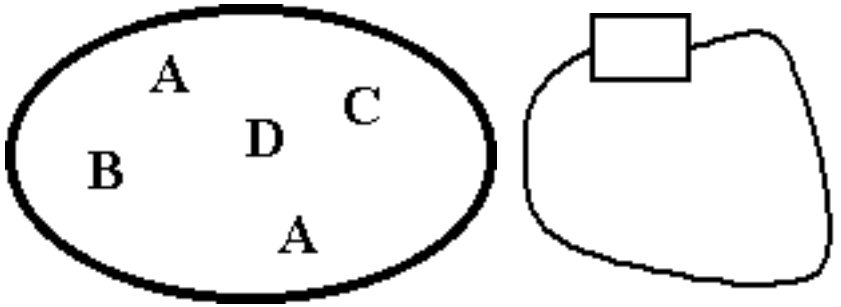
- Creiamo una collezione e inseriamo degli elementi (non facciamo assunzioni su ordine e ripetizioni dei suoi elementi)

```
Collection<Item> coll = new ...;
coll.add(...);
...
```



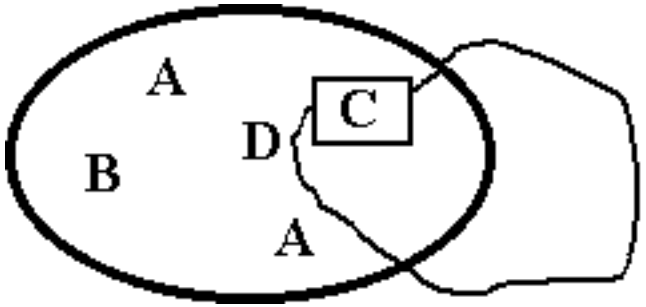
Interpretazione grafica - 2

- Creiamo un iteratore sulla collezione **coll**
  `Iterator<Item> it = coll.iterator( );`
- Lo rappresentiamo come un “sacchetto” con una “finestra”
  - la finestra contiene l’ultimo elemento visitato
  - Il sacchetto quelli già visitati



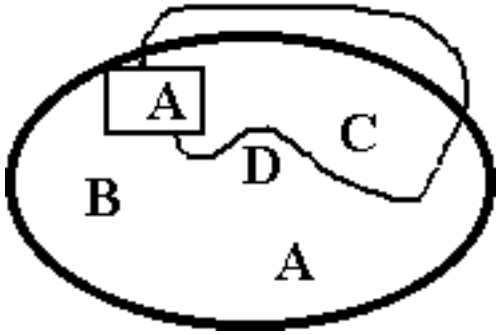
Interpretazione grafica - 3

- Invoco **it.next( )**: restituisce, per esempio, l'elemento **C**
- Graficamente, la finestra si sposta su **C**



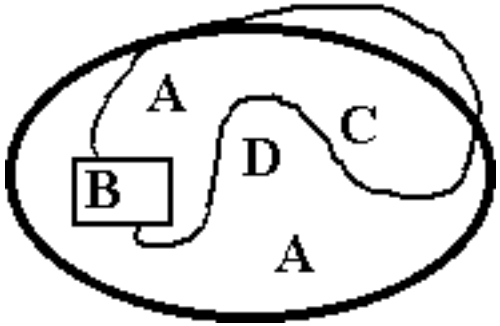
Interpretazione grafica - 4

- Invoco nuovamente **it.next( )**: ora restituisce l'elemento **A**
- Graficamente, la finestra si sposta su **A**, mentre l'elemento **C** viene messo nel sacchetto per non essere più considerato



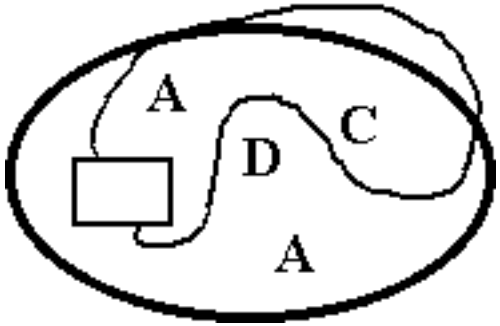
Interpretazione grafica - 5

- **it.next( )** restituisce **B**
- **it.hasNext( )** restituisce **true** perché c'è almeno un elemento "fuori dal sacchetto"



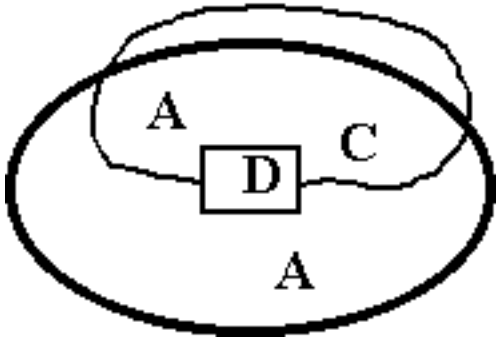
Interpretazione grafica - 6

- **it.remove( )** cancella dalla collezione l'elemento nella finestra, cioè **B** (l'ultimo visitato)
- Un invocazione di **it.remove( )** quando la finestra è vuota lancia una **IllegalStateException**



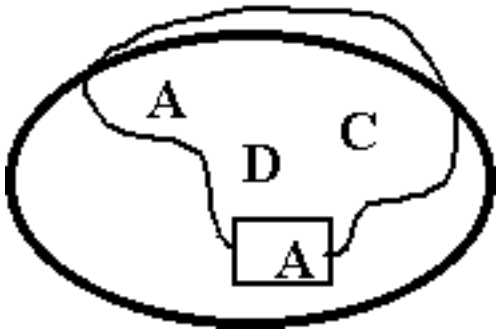
Interpretazione grafica - 7

- **it.next( )** restituisce **D**



Interpretazione grafica - 8

- **it.next( )** restituisce **A**
- Ora **it.hasNext( )** restituisce **false** perché non ci sono altri elementi da visitare
- Se eseguo ancora **it.next( )** viene lanciata una **NoSuchElementException**



Riassumendo: uso di iteratore

- Con successive chiamate di **next( )** si visitano tutti gli elementi della collezione esattamente una volta
- **next( )** lancia una **NoSuchElementException** esattamente quando **hasNext( )** restituisce **false**
- L'ordine nel quale vengono restituiti gli elementi dipende dall'implementazione dell'iteratore
  - una collezione può avere più iteratori, che usano ordini diversi
  - per le collezioni lineari (come **List**) l'iteratore default rispetta l'ordine
- Si possono attivare più iteratori simultaneamente su una collezione
- Se invoco la **remove( )** senza aver chiamato prima **next( )** si lancia una **IllegalStateException( )**
- Se la collezione viene modificata durante l'iterazione di solito viene invocata una **ConcurrentModificationException**

Iterazione, astrendo dalla collezione

- Java fornisce meccanismi per realizzare, tramite gli iteratori, algoritmi applicabili a qualunque tipo di collezione
- Creazione di iteratore default su collezione con il metodo **Iterator<E> iterator( )**
  - definito nell'interfaccia **Iterable<E>** che è estesa da **Collection<E>**
- Esempio: stampa degli elementi di una quasiasi collezione

```
public static <E> void print(Collection<E> coll) {
   Iterator<E> it = coll.iterator();
   while (it.hasNext( ))  // finché ci sono elementi
      System.out.println(it.next( ));
}
```

Il comando **for-each** (*enhanced* ***for***)

- Da Java 5.0: consente l'iterazione su tutti gli elementi di un **array** o di una **collezione** (o di un oggetto che implementa **Iterable<E>**)

```
Iterable<E> coll = ...;
for (E elem : coll)  System.out.println(elem);
// equivalente a
Iterable<E> coll = ...;
Iterator<E> it = coll.iterator( );
while (it.hasNext( )) System.out.println(it.next( ));

E[ ] arr = ...;
for (E elem : arr)  System.out.println(elem);
// equivalente a
E[ ] arr = ...;
   for (int i = 0; i < arr.size( ); i++)
       System.out.println(arr[i]);
```

Modifiche concorrenti

- Contratto con iteratore: la collezione può essere modificata solo attraverso l'iteratore (con **remove**)
- Se la collezione viene modificata durante l'iterazione, di solito viene lanciata una **ConcurrentModificationException**
- Esempio

```
List<Integer> lst = new Vector<Integer>( );
lst.add(1); // collezione con un solo elemento
Iterator<Integer> it = lst.iterator( );
System.out.println(it.next( )); // stampa 1
lst.add(4); // modifica esterna all'iteratore!!!
it.next( ); // lancia ConcurrentModificationException
```

Iteratori e ordine superiore

- L'uso degli iteratori permette di separare la **generazione** degli elementi di una collezione (ad es. intestazione di **for-each**) dalle operazioni che si fanno su di essi (corpo di **for-each**)
- Questo si realizza facilmente con normale astrazione procedurale in linguaggi (funzionali) nei quali le procedure sono "cittadini di prima classe", cioè valori come tutti gli altri
  - possono essere passate come parametri ad altre procedure
    - esempio: **map** di Ocaml
- Da Java 8 (2014) è possible anche in Java è possible definire funzioni anonime (**lambda expressions**) da passare come parametri a metodi quali filter, map, ecc…
  - questa possibilità si traduce in realtà in un uso nascosto di un iteratore (come nel caso di for-each)

Specifica di iteratori

- Abbiamo visto come **si usa** un iteratore associato a una collezione
- Vediamo come si **specificano** e come si **implementano**
- Vediamo anche come si definisce un iteratore “stand alone”, che genera elementi senza essere associato a una collezione
- Useremo **generatore** per iteratore non associato a una collezione
- L’implementazione farà uso di **classi interne**

Specifica di iteratore per `IntSet`

```
public class IntSet implements Iterable<Integer> {

public Iterator<Integer> iterator( );
  // REQUIRES: this non deve essere modificato
  // finche' il generatore e' in uso
  // EFFECTS: ritorna un iteratore che produrra' tutti
  // gli elementi di this (come Integers) ciascuno una
  // sola volta, in ordine arbitrario
}
```

- La clausola REQUIRES impone condizioni sul codice che utilizza il generatore
  - –tipica degli iteratori su tipi di dati modificabili
- Dato che la classe implementa **Iterable<E>** si può usare **for-each**

Specifica di generatore stand alone

```
public class Primes implements Iterable<Integer> {
  public Iterator<Integer> iterator( )
  // EFFECTS: ritorna un generatore che produrra' tutti
  // i numeri primi (come Integers), ciascuno una
  // sola volta, in ordine crescente
}
```

- Un tipo di dato può avere anche più iteratori, quello restituito dal metodo **iterator( )** è il "default"
- In questo caso il limite al numero di iterazioni deve essere imposto dall'esterno
  - il generatore può produrre infiniti elementi

Uso di iteratori: stampa primi

```
public class Primes implements Iterable<Integer> {
  public Iterator<Integer> iterator( );
  // EFFECTS: ritorna un iteratore che produrrà tutti i numeri
  // primi (come Integers) ciascuno una sola volta, in ordine
  //crescente
}

public static void printPrimes (int m) {
  // EFFECTS: stampa tutti i numeri primi minori o uguali a m
  // su System.out
  for (Integer p : new Primes( )){
    if (p > m) return; // forza la terminazione
    System.out.println("The next prime is: " + p);
  }
}
```

Uso di iteratori: massimo

- Essendo oggetti, gli iteratori possono essere passati come argomento a metodi che astraggono da dove vengono gli argomenti sui quali lavorano
  - **max** funziona per qualunque iteratore di interi

```
public static int max (Iterator<Integer> g)
  throws EmptyException, NullPointerException {
   // EFFECTS: se g e' null solleva NullPointerException; se g e'
   // vuoto solleva EmptyException, altrimenti visita tutti gli
   // elementi di g e restituisce il massimo intero in g
   try { int m = g.next( );
         while (g.hasNext( )) { int x = g.next( );
                                if (m < x) m = x; }
         return m;
   } catch (NoSuchElementException e){ throw new EmptyException("max"); }
}
```

Implementazione degli iteratori

- Gli iteratori/generatori sono oggetti che hanno come tipo un sotto-tipo di `Iterator`
  - istanze di una classe γ che "implementa" l'interfaccia `Iterator`
- Un metodo α (stand alone o associato a un tipo astratto) ritorna l'iteratore istanza di γ. Tipicamente α è **iterator( )**
  - γ deve essere contenuta nella stessa classe che contiene α
    - ✓ dall'esterno si deve poter vedere solo il metodo α
    - ✓ non la classe γ che definisce l'iteratore
- La classe γ può essere contenuta nella classe che contiene α
    - ✓ come classe interna privata
- Dall'esterno gli iteratori sono visti come oggetti di tipo `Iterator`: il sotto-tipo γ non è visibile

Classi interne / annidate

- Una classe γ dichiarata come membro all'interno di una classe α può essere
  - static (di proprietà della classe α)
  - di istanza (di proprietà degli oggetti istanze di α)
- Se γ è static, come sempre non può accedere direttamente alle variabili di istanza e ai metodi di istanza di α
  - le classi che definiscono i generatori sono definite quasi sempre come classi interne, statiche o di istanza

Classi interne: semantica

- La presenza di classi interne richiede la presenza di un ambiente di classi
  - all'interno delle descrizioni di classi
  - all'interno degli oggetti (per classi interne non static)
  - vanno modificate di conseguenza anche tutte le regole che accedono i nomi

Implementazione iteratori: IntSet

```
public class IntSet implements Iterable<Integer> {

   private int[] a;
   private int size;

   public IntSet(int capacity) { ... }
   public boolean add(int elem) throws FullSetException { ... }
   public boolean contains(int elem) { ... }

   public Iterator<Integer> iterator( ) {
        return new IntSetIterator();
   }

   // INNER CLASS (CLASSE INTERNA)
   private class IntSetIterator implements Iterator<Integer> {
        ... (SEGUE) ...
   }
}
```

Implementazione iteratori: IntSet

```
public class IntSet implements Iterable<Integer> {

   ...

   // INNER CLASS (CLASSE INTERNA)
   private class IntSetIterator implements Iterator<Integer> {
        private int curr=0;

        public boolean hasNext() { return (curr<size); }

        public Integer next() {
               if (curr>=size) throw new NoSuchElementException();
               return a[curr++];
        }
        public void remove( ) {
               throw new UnsupportedOperationException( );
        }
   }
}
```

Implementazione iteratori: IntSet

- Una classe interna usata una volta sola nella classe che la contiene può essere definite anche come classe anonima (anonymous class)
- Nel fare la **new** si indica il supertipo della classe (che può essere anche dato da un'interfaccia) e direttamente (inline) il codice della classe, senza darle un nome

Implementazione iteratori: IntSet

```
public class IntSet implements Iterable<Integer> {
   ...

   public Iterator<Integer> iterator( ) {
     // ANONYMOUS (INNER) CLASS
     return new Iterator<Integer>() {
        private int curr=0;
        public boolean hasNext() { return (curr<size); }
        public Integer next() {
                if (curr>=size) throw new NoSuchElementException();
                return a[curr++];
        }
        public void remove( ) {
                throw new UnsupportedOperationException( );
        }
     };
   }

   ...
}
```

Implementazione iteratori: Primes

```
public class Primes implements Iterable<Integer> {
  public Iterator<Integer> iterator( ) { return new PrimeGen( ); }
    // EFFECTS: ritorna un generatore che produrra' tutti i numeri primi
    // (come Integers) ciascuno una sola volta, in ordine crescente
    private static class PrimeGen implements Iterator<Integer> {
      // class interna statica
      private List<Integer> ps; // primi gia' dati
      private int p; // prossimo candidato alla generazione
      PrimeGen( ) { p = 2; ps = new ArrayList<Integer>( ); } // costruttore
      public boolean hasNext( ) { return true; }
      public Integer next( ) {
        if (p == 2) { p = 3; ps.add(2); return new Integer(2); }
          for (int n = p; true; n = n + 2)
            for (int i = 0; i < ps.size( ); i++) {
              int e1 = ps.get(i);
              if (n%e1 == 0) break; // non e' primo
              if (e1*e1 > n) { ps.add(n); p = n + 2; return n; }
            }
        }
      public void remove( ) { throw new UnsupportedOperationException( ); }
  }
}
```

Considerazioni sugli iteratori

- In molti tipi di dato astratti (collezioni) gli iteratori sono un componente essenziale
  - supportano l'astrazione via specifica
  - portano a programmi efficienti in tempo e spazio
  - sono facili da usare
  - non distruggono la collezione
  - ce ne possono essere più d'uno
- Se il tipo di dati astratto è modificabile ci dovrebbe sempre essere il vincolo sulla non modificabilità della collezione durante l'uso dell'iteratore
  - altrimenti è molto difficile specificarne il comportamento previsto
  - in alcuni casi può essere utile combinare generazioni e modifiche

Livelli di astrazione

Gli iteratori (in general il JCF) mostrano bene che cosa significhi lavorare per livelli di astrazione:

```
public void printSet (IntSet set) {
  for (Integer i : set) System.out.println(i);
}
```

enhanced for (for-each) su IntSet

---------- ASTRAE DA ----------

hasNext(), next(), … definizione di un iteratore

---------- ASTRAE DA ----------

int[] a; int size; rappresentazione dell'insieme

---------- ASTRAE DA ----------

heap, garbage collection, … gestione della memoria nella JVM

---------- ASTRAE DA ----------

malloc/new, free/delete, … implementazione (in C++) della JVM

Generare e modificare

- Programma che esegue task in attesa su una coda di task

```
Iterator<Task> g = q.allTasks( );
while (g.hasNext( )) {
   Task t = g.next( );
   // esecuzione di t
   // se t genera un nuovo task nt, viene messo
   // in coda facendo q.enqueue(nt)
}
```

- Casi come questo sono molto rari
- L'interfaccia **ListIterator<E>** definisce iteratori più ricchi, perché assumono di lavorare su una collezione *doubly linked*
  - permettono di spostarsi in avanti e indietro (**next( )** e **previous( )**)
  - permettono di modificare la lista con **add(E e)** e **set(E e)** oltre a **remove( )**

Sulla modificabilità

- Due livelli: modifica di collezione e modifica di oggetti
- Le collezioni del **JCF** sono modificabili
- Si possono trasformare in non modificabili con il metodo `unmodifiableCollection` messo a disposizione dalla classe `Collections`

```
public static <T> Collection<T>
    unmodifiableCollection(Collection<? extends T> c)
```

- Anche se la collezione non è modificabile, se il tipo base della collezione è modificabile, si può modificare l'oggetto restituito dall'iteratore. Questo non modifica la struttura della collezione, ma il suo contenuto
- Infatti gli iteratori del **JCF** restituiscono gli elementi della collezione, non una copia

Per esercizio: iteratore per array

- Gli array possono essere visitati con il **for-each**, ma non hanno un iteratore default
- Si realizzi la classe ArrayIterator che rappresenta un iteratore associato a un vettore di interi: in particolare
  - la classe ArrayIterator implementa l'interfaccia Iterator<Integer>
  - il costruttore della classe ha come unico parametro l'array che contiene gli elementi sui quali iterare
  - implementa tutti i metodi di Iterator<Integer> e in particolare public boolean hasNext( ) e public Integer next( )
- Suggerimento: per mantenere traccia dell'attuale elemento dell'iteratore si utilizzi un campo intero che rappresenta l'indice del vettore
- Si discuta un'eventuale realizzazione della remove( )
- Si renda generica la classe, in modo da realizzare un iteratore su un generico array di tipo base T

Per esercizio: iteratore inverso per Vector

- L'iteratore standard della classe **Vector<T>** scandisce gli elementi dalla posizione **0** fino alla posizione **size( ) – 1**
- Si scriva la classe **RevVector<T>** che estende **Vector<T>** e ridefinisce il solo metodo **iterator( )**, restituendo un oggetto della classe **RevVectIterator<T>**
- Si definisca la classe **RevVectIterator<T>** che implementa **Iterator<T>** e realizza un iteratore che scandisce gli elementi di un vettore in ordine inverso a quello standard. Si discutano più opzioni: (1) classe top-level, classe interna (2) statica / (3) non statica, in particolare rispetto alla visibilità e ai parametri da passare al costruttore
- Si scriva la classe **PrintCollection** che contiene il metodo statico

  public static <T> printCollection(Collection<T> coll)
- che stampa tutti gli elementi della collezione **coll** usando un iteratore
  - si costruisca nel **main** un oggetto di **Vector** e uno di **RevVector**, riempiendoli con gli stessi elementi, e si invochi **printCollection** su entrambi, per verificare l'ordine nel quale vengono stampati gli elementi