AA 2016-2017

14. 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)

L'interfaccia `Collection<E>`

- Definisce operazioni basiche su collezioni, senza assunzioni su struttura/modificabilità/duplicati…
- Modifiers **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( ); }
```

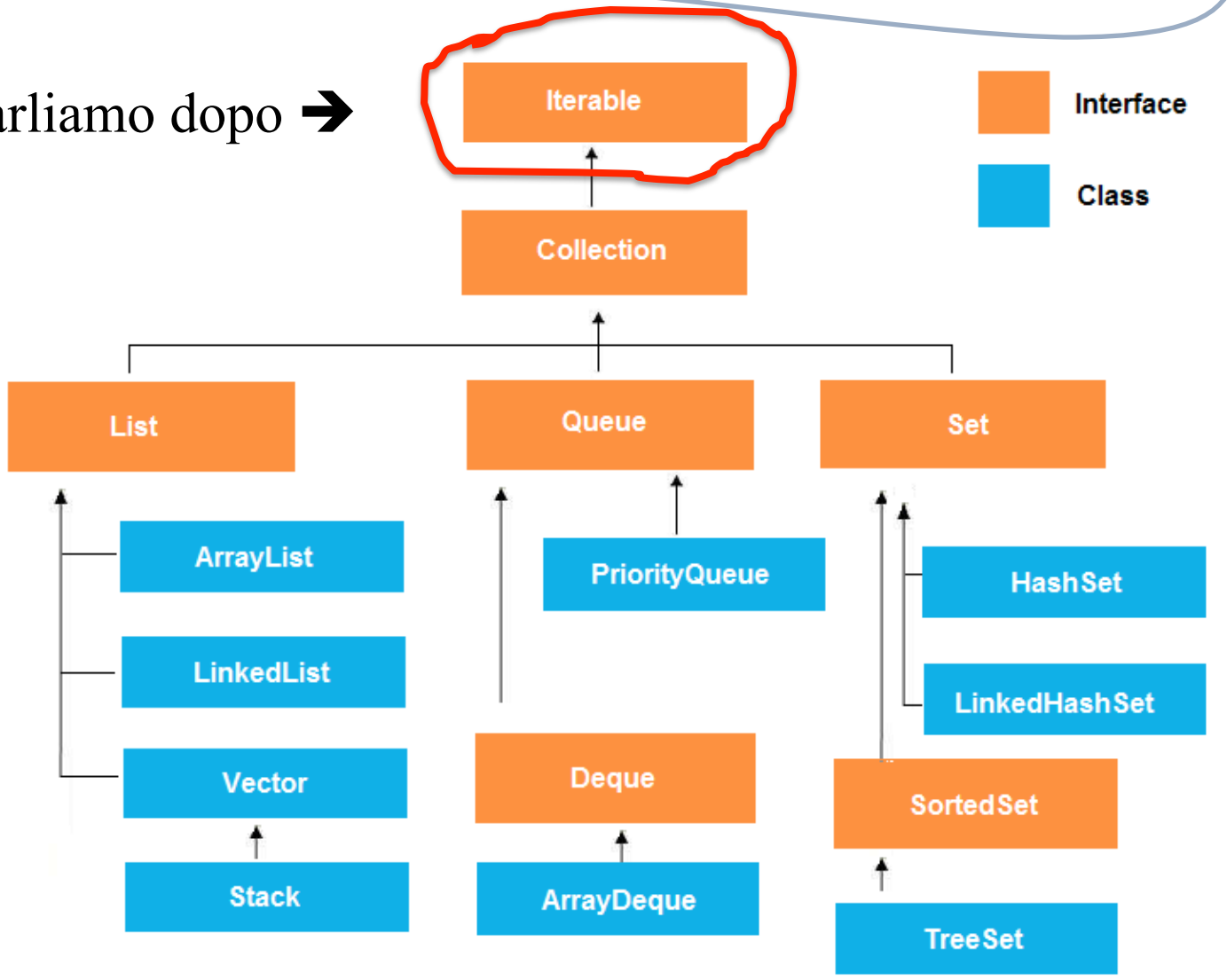
- Observers: `contains(o), equals(o), isEmpty( ), size( ), toArray()`
- Accesso agli elementi con `iterator( )` (v. dopo)

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(o), remove(int index),get(index), set(index, el)`
- **`Queue<E>`**: supporta politica FIFO
  - **`Deque<E>`**: "double ended queue", "deck". 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`)

JCF: parte della gerarchia

Ne parliamo dopo ➔



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

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 o una lista si può usare un **for**

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

- Infatti, per array e vettori sappiamo
  - la dimensione: quanti elementi contengono (**length** o **size( )**)
  - 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( );
}
```

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

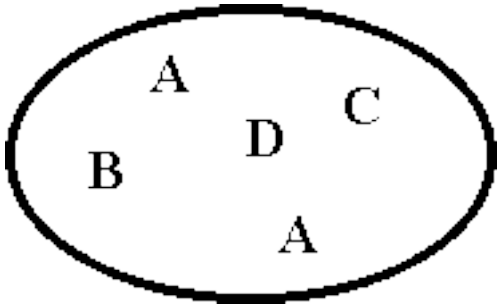
```
// creo un iteratore sulla collezione
Iterator<Integer> it = myIntCollection.iterator( );
while (it.hasNext( )) {  // finché ci sono elementi
    int x = it.next( );  // prendo il prossimo
    // usa  x
}
```

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. */
}
```

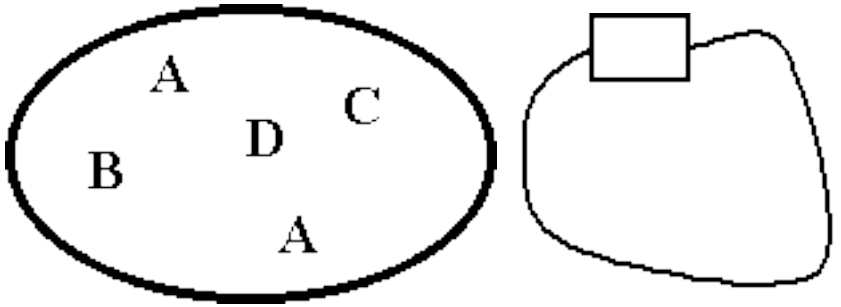
Interpretazione grafica - 1

- Creiamo una collezione e inseriamo degli elementi (non facciamo assunzioni su ordine e ripetizioni dei suoi elementi)
  Collection<Item> coll = new ...;
  coll.add(...);
  ...



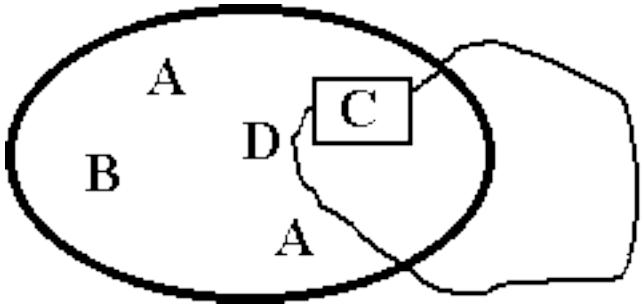
Interpretazione grafica - 2

- 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



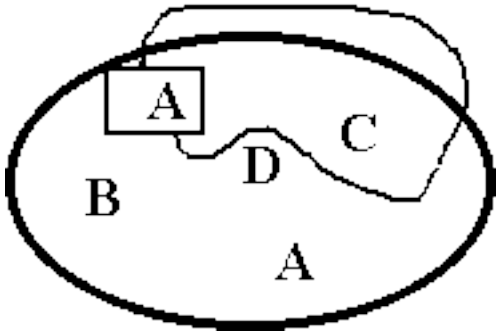
Interpretazione grafica - 3

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



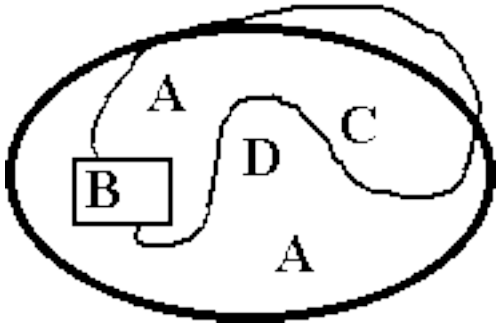
Interpretazione grafica - 4

- 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



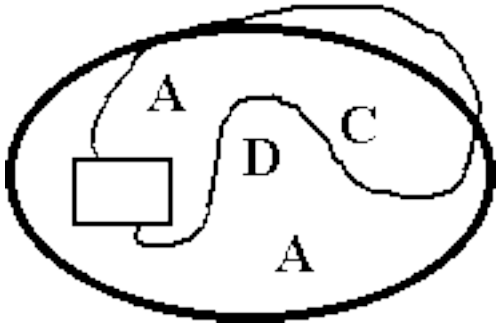
Interpretazione grafica - 5

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



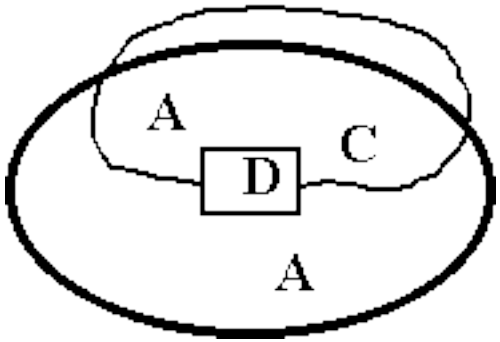
Interpretazione grafica - 6

- **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**



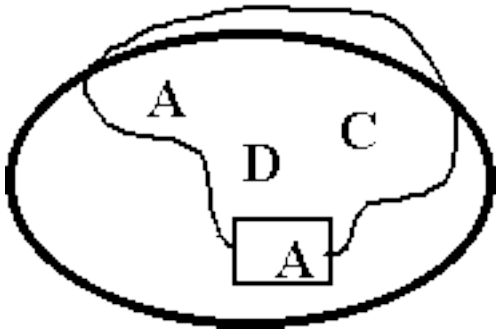
Interpretazione grafica - 7

- **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 contemporaneamente 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
  - il generatore è una procedura che ha come parametro la procedura che codifica l'azione da eseguire sugli elementi della collezione
  - il generatore (parametrico) è una operazione del tipo astratto
    - ✓ esempio: **map** di OCaml

Specifica di iteratori

- Abbiamo visto come **si usa** un iteratore associato ad una collezione
- Vediamo come si **specificano** e come si **implementano**
- Consideriamo la classe **IntSet** del libro di Liskov, ma aggiornandola rispetto a Java 5.0
- Vediamo anche come si può definire un iteratore "stand alone", che genera elementi senza essere associato ad 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> {
// come prima piu'
  public Iterator<Integer> iterator( );
  // REQUIRES: this non deve essere modificato
  // finché 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
- Indicando che la classe implementa **Iterable<E>** si può usare il **for-each**

Specifica di generatore stand alone


```
public class Primes implements Iterator<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 Iterator<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) {
  // MODIFIES: System.out
  // 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 così astraggono da dove provengono 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 sottotipo 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 nello stesso modulo che contiene α
    - ✓ dall'esterno del modulo si deve poter vedere solo il metodo α (con la sua specifica)
    - ✓ non la classe γ che definisce l'iteratore
- La classe γ deve avere una visibilità limitata al package che contiene α
  - oppure può essere contenuta nella classe che contiene α
    - ✓ come classe interna privata
- Dall'esterno gli iteratori sono visti come oggetti di tipo `Iterator`: il sottotipo 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: 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( ); }
     }
}
```

Classi interne e iteratori

- Le classi i cui oggetti sono iteratori definiscono comunque dei tipi astratti
  - sottotipi di `Iterator`
- In quanto tali devono essere dotati di
  - una invariante di rappresentazione
  - una funzione di astrazione
    - ✓dobbiamo sapere cosa sono gli stati astratti
    - ✓per tutti gli iteratori, lo stato astratto è
      - la sequenza di elementi che devono ancora essere generati
    - ✓la funzione di astrazione mappa la rappresentazione su tale sequenza

Generatore di numeri primi: FA


```
public class Primes implements Iterable<Integer> {
  public Iterator<Integer> iterator( ) { return new PrimeGen( ); }
  private static class PrimeGen implements Iterator<Integer> {
  // class interna statica
     private List<Integer> ps; // primi già dati
      private int p; // prossimo candidato alla generazione

   // la funzione di astrazione
   // α(c) = [ p1, p2, ...] tale che
   // ogni pi e' un Integer, e' primo ed e' < c.p,
   // tutti i numeri primi < c.p occorrono nella
   //       sequenza, e
   // pi > pj per tutti gli i > j > 1
```

Generatore di numeri primi: IR


```
public class Primes implements Iterable<Integer> {
  public Iterator<Integer> iterator( ) { return new PrimeGen( ); }
  private static class PrimeGen implements Iterator<Integer> {
  // class interna statica
  private List<Integer> ps; // primi gia' dati
  private int p; // prossimo candidato alla generazione

   // l'invariante di rappresentazione
   // I(c) = c.ps != null,
   // tutti gli elementi di c.ps sono primi,
   // sono ordinati in modo crescente e
   // contengono tutti i primi < c.p e >= 2
```

Conclusioni 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

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.enq(nt)
}
```

- Casi come questo sono molto rari
- L'interfaccia **ListIterator<E>** definisce iteratori più ricchi, perché assumono di lavorare su una collezione lineare *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 che con **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

```
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

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 Integer[ ] a 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

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 nuova 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