PROGRAMMAZIONE 2
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

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 algoritmi testati
  - Efficienza implementazioni
  - Interoperabilità
  - Riuso del software
- Le userete nel progetto di Java

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( );
}
```

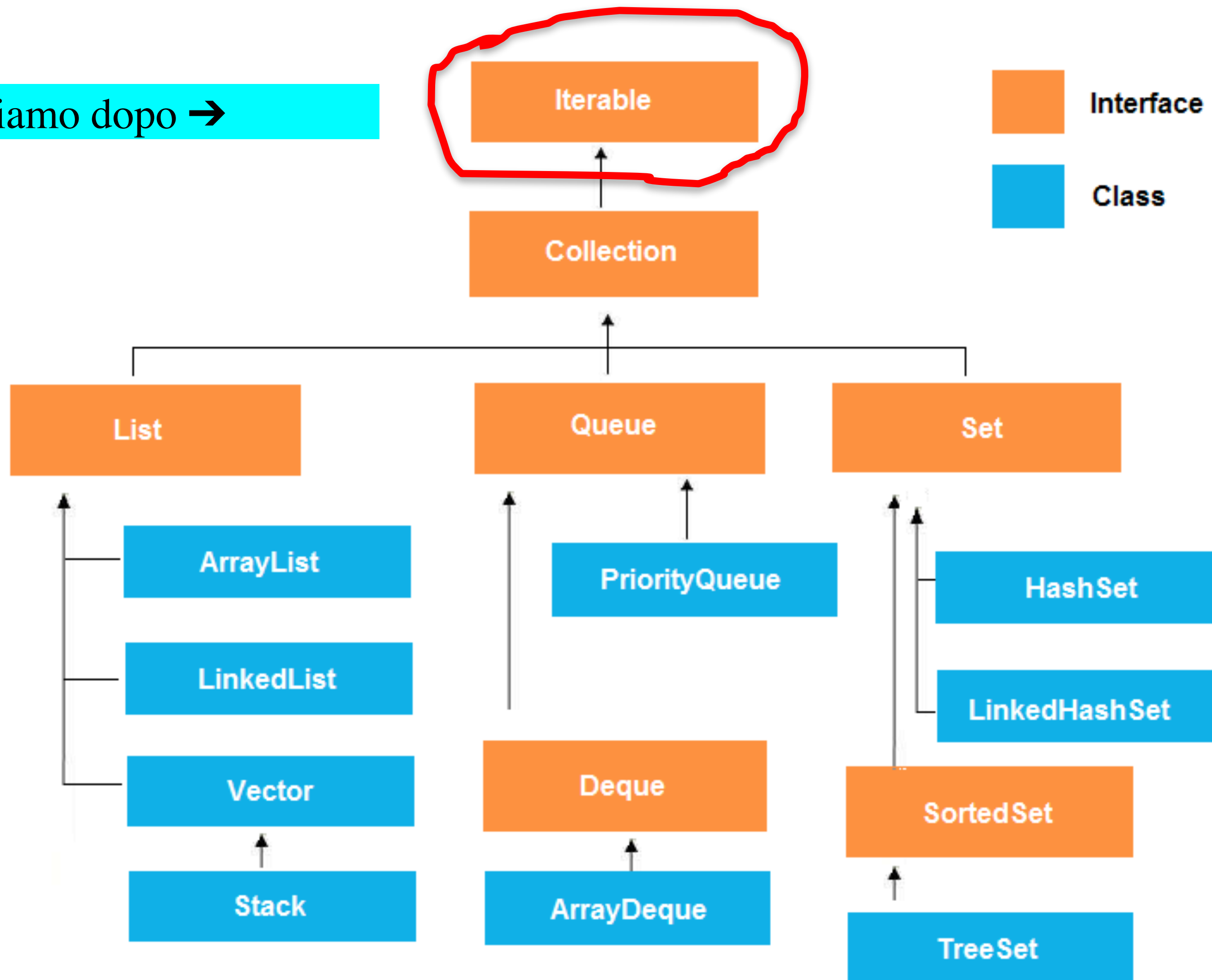
- Observers: `contains(o)`, `equals(o)`, `isEmpty( )`, `size( )`, `toArray()`
- Accesso agli elementi con `iterator( )` (vedi dopo)

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", "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));
```

Iterazione su collezioni: motivi

- Possiamo iterare in modo astratto (senza conoscere la rappresentazione) dato che per array e vettori sappiamo
- La dimensione: quanti elementi contengono (**`length`** o **`size( )`**)
- come accedere in modo diretto ad ogni elemento con un indice
- Come possiamo iterare in modo astratto per un TDA?

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( );
        }
```

Specifica 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. */
 }
```

...e il loro uso (iterazione astratta)

```
// creo un iteratore sulla collezione

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

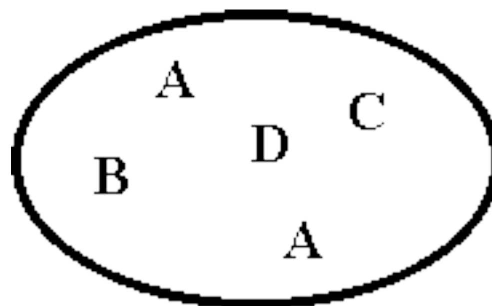
Poi vedremo che il metodo `iterator` restituisce un generatore sulla collezione

Interpretazione grafica - 1

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

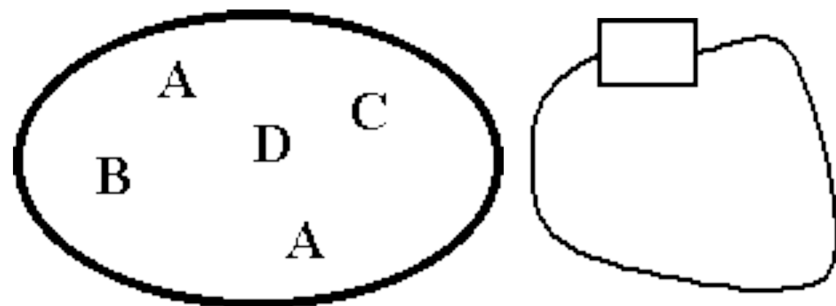
```
Collection<Item> coll = new ...;
coll.add(...);
```

...



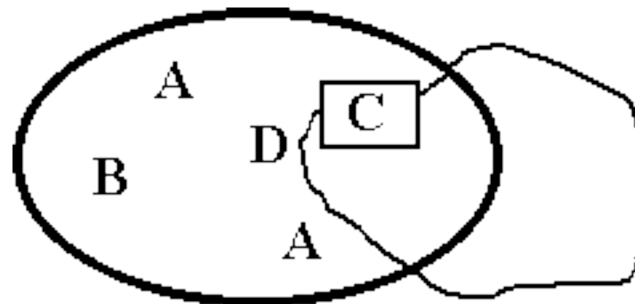
Interpretazione grafica - 2

- 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



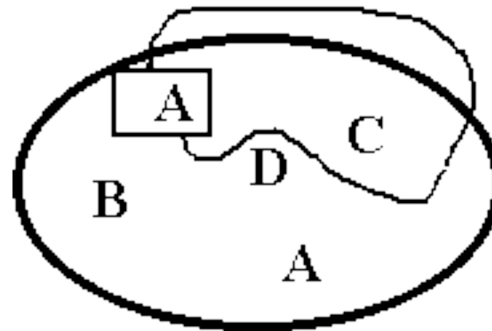
Interpretazione grafica - 3

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



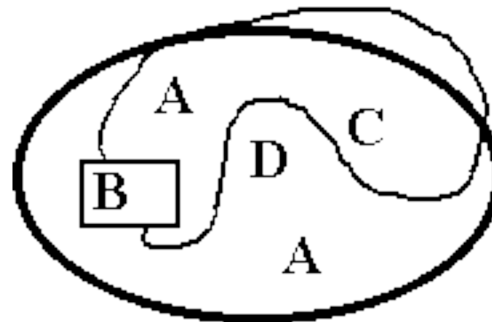
Interpretazione grafica - 4

- 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



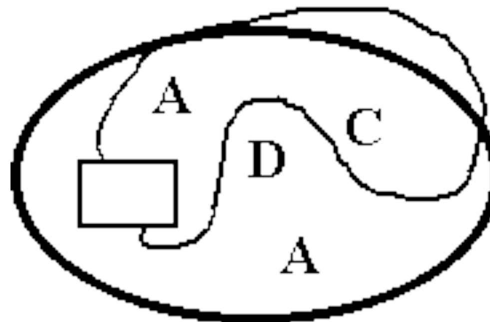
Interpretazione grafica - 5

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



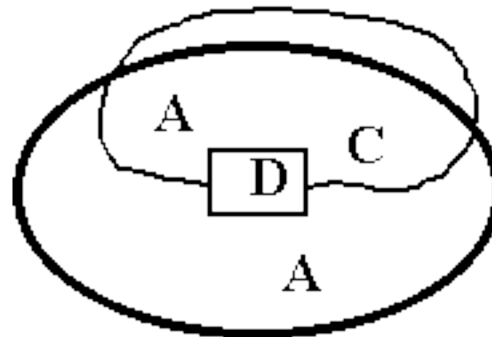
Interpretazione grafica - 6

- **`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`**



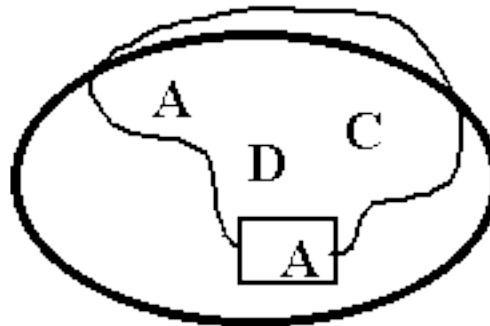
Interpretazione grafica - 7

- `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 di 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, astraendo dalla collezione

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

Specifica di iteratori

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

Specifica: un iteratore per **IntSet**

```
public class IntSet implements  Iterable<Integer> {
// specifica standard +
  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<Integer>` si può usare `for-each`

Specifica: un iteratore 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 quello di “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 generatore che produrra' 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);
  }
}
```

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 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 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

Implementazione: PrimeGen

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

Implementazione: PrimeGen

```
private static class PrimeGen implements Iterator<Integer> {
    private List<Integer> ps; // primi gia' dati
    private int p; // prossimo candidato alla generazione
    public PrimeGen( ) {p = 2; ps = new ArrayList<Integer>( ); }
    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 TDA
  - sotto-tipi 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 FA mappa la rappresentazione su tale sequenza

Generatore di numeri primi: FA

```
private static class PrimeGen
              implements Iterator<Integer> {
 private List<Integer> ps; // primi gia' 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

```
private static class PrimeGen
                    implements Iterator<Integer> {
 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 TDA (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 TDA 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

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

- Completare l'implementazione della classe **`IntSet`** (implementare il generatore ed il corrispondente metodo **`iterator`**)
- Classe interna privata
- Gli elementi possono essere generati in ordine arbitrario !!!!
- Dare invariante di rappresentazione e funzione di astrazione
- Ragionare sulla correttezza (se il vettore che implementa l'insieme avesse elementi duplicati?)
- Scrivere una classe che contiene due metodi statici che prendono un **`IntSet`** e calcolano il min e la somma dei valori dell'insieme (si può usare **`for-each`**) iterazione astratta