

Astrazioni sul controllo

Specifica e Implementazione degli Iteratori

Nuove iterazioni

- Definendo un nuovo tipo come collezione di oggetti (p. es., set) si vorrebbe disporre anche di un'operazione che consenta cicli (iterazioni)
 - Es.: gli elementi possono essere stampati, oppure sommati tra loro, oppure confrontati per cercare il più piccolo o il più grande, ecc.
 - Es. IntSet: l'utilizzatore ha bisogno di verificare se tutti gli elementi sono positivi
- Se la collezione è un Vector, un array o un ArrayList, è facile
 - Es. stampare tutti gli elementi del vettore vect, dal primo all'ultimo:
for (int i = 0; i < vect.size(); i++) System.out.println(vect.get(i));
 - array e vettori: organizzazione lineare degli elementi, e possiamo accedere ad un qualunque elemento usando un indice
- Con altri contenitori/collezioni? Con IntSet? non c'è get o size
 - Information hiding non consente (giustamente!) l'accesso diretto al rep di un contenitore! Non possiamo scrivere, se set è un IntSet:
for (int i = 0; i < set.els.size(); i++) System.out.println(set.els.get(i));

Iterare su collezioni

- **Soluzione poco generale:** scrivere metodi in IntSet per stampare e per verificare se elementi positivi
 - come fare a prevedere in anticipo tutti gli usi di IntSet? Es. “calcolare la somma di tutti gli elementi”
- **Soluzione inefficiente :** IntSet ha un metodo che restituisce un array con tutti gli elementi
 - si consuma spazio e tempo di memoria (es. ricerca che si interrompe al primo elemento...). A volte però è utile.
- **Soluzione inefficiente:** fornire anche IntSet di size e di get
for (int i = 0; i < set.size(); i++)

System.out.println(set.get(i));

- con IntSet funzionerebbe bene finché si usa Vector. Ma se poi si cambiasse implementazione? se fosse con lista... Su molte strutture dati accesso casuale a i-esimo elemento può essere molto inefficiente! Es. con linked list, occorre sempre scorrere gli elementi precedenti a quello in posizione i: numero accessi alla lista tramite la get(i): $1 + 2 + 3 + \dots + n = n(n+1)/2$, con $n = \text{set.size}()$.

Iteratori

- **Soluzione generale:** introdurre oggetti detti **iteratori** in grado di “spostarsi” sugli elementi dei contenitori
 - Associare a ogni contenitore un iteratore
 - Iteratore rappresenta un'astrazione del concetto di “puntatore a un elemento del contenitore” e permette di scorrere il contenitore senza bisogno di conoscere l'effettivo tipo degli elementi.
 - Es. IntSet: oggetto “iteratore su IntSet” con metodi: **next()** per restituire l'elemento su cui si è posizionati e muoversi su quello successivo; **hasNext()** per verificare se siamo sull'ultimo elemento.

```
public static boolean insiemePositivo(IntSet set){  
    “genera iteratore itr per set (posizionato al primo elemento)”;  
    while (itr.hasNext( ))  
        if (itr.next()) <0) return false;  
    return true;}  
}
```

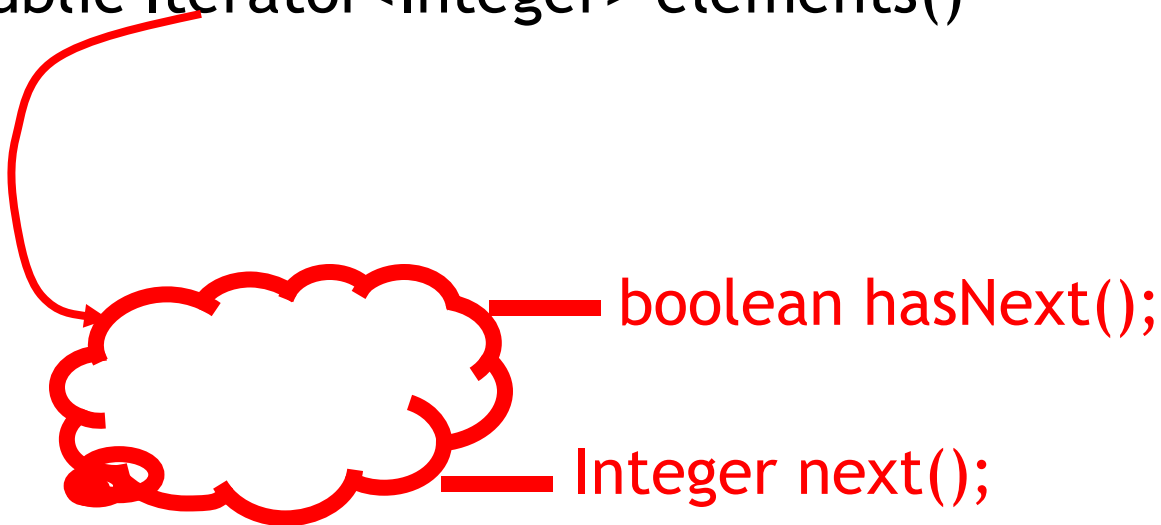
Da Eckel, Thinking in Java:

- There's not much you can do with the Java **Iterator** except:
 - Ask a container to hand you an **Iterator** using a method called **iterator()**. This **Iterator** will be ready to return the first element in the sequence on your first call to its **next()** method.
 - Get the next object in the sequence with **next()**.
 - See if there *are* any more objects in the sequence with **hasNext()**.
 - Remove the last element returned by the iterator with **remove()**

Iteratori in Java

- Una collezione deve definire (almeno) un metodo che ritorna un oggetto di tipo Iterator
- Nome standard: iterator() ma sarebbero possibili anche altri nomi (ad esempio, elements())

```
//@ ensures (* ritorna un oggetto che  
//@ consente di iterare su IntSet *);  
public Iterator<Integer> elements()
```



Interface Iterator

- In Java, una interfaccia è una dichiarazione di quali metodi pubblici deve possedere una classe che “implementa” l’interfaccia
- Per standardizzare l’uso degli iteratori, in Java c’è interfaccia Iterator in java.util (dove si trova anche la run-time exception NoSuchElementException)

```
public interface Iterator<E> {
```

```
    //@ensures (* \result è true se e solo se esiste un prossimo elemento *);  
    public boolean hasNext ( );
```

```
    //@ensures (* fornisce il prox elemento se esiste *);  
    //@ signals (NoSuchElementException e)  
    //          (*non esiste un prossimo elem. *);  
    public E next() throws NoSuchElementException;
```

```
    //@ensures (* rimuove ultimo elemento ritornato da next() *);  
    public void remove ( );
```

```
        //OPZIONALE, nel senso che si può implementare senza rimozione
```

```
}
```

- NB: Per compatibilità con versioni precedenti di Java esiste versione “raw” del tipo Iterator che non ha parametro e si riferisce implicitamente a elementi di tipo Object
 - simile a quanto avviene con i contenitori ArrayList e Vector
 - ciò accade in realtà per tutti i tipi generici del linguaggio

Uso iteratore per IntSet

```
public static boolean stampa(IntSet set){  
    for (Iterator<Integer> itr = set.elements(); itr.hasNext( );)  
        System.out.println(itr.next());  
}
```

```
public static boolean insiemePositivo(IntSet set){  
    for (Iterator<Integer> itr = set.elements(); itr.hasNext( );)  
        if (itr.next()) <0) return false;  
    return true;}  

```

In alternativa, funzione può essere definita in base solo a iteratore!

Iterare su collezioni qualunque

- Vantaggio: molti algoritmi (es. ordinamento, ricerca, ...) possono essere descritti solo in base a iteratori e alcuni metodi standard, indipendentemente dal tipo degli elementi!
- È potente astrazione sul controllo: “the true power of the **Iterator**: the ability to separate the operation of traversing a sequence from the underlying structure of that sequence”.
- Esempi: stampa; ricerca sequenziale, con confronto con equals()

```
class Iterazioni {  
    public static void printAll(Iterator<E> itr) {  
        while(itr.hasNext())  
            System.out.println(itr.next()); //NB:conv. automatica a String  
    }  
    //@ ensures (* \result == o è presente nella collezione associata all'iteratore itr *);  
    public static boolean cerca(Iterator<E> itr, <E> o){  
        while (itr.hasNext( )) if (o.equals(itr.next())) return true;  
        return false;}}
```

Generatori e iteratori; remove

- Nomenclatura standard:
 - **iteratore** è oggetto che consente di muoversi su collezioni
 - Interfaccia Iterator è in java.util e fornisce anche remove() (opzionale)
- Nomenclatura Liskov:
 - **generatore** è oggetto che consente di muoversi su collezioni
 - **iteratore** è un metodo che restituisce un generatore
 - Interfaccia Iterator non ha remove()

Specifica: metodi iteratori per Poly e IntSet

```
public class Poly {  
  //come visto prima, e in più:  
  public Iterator<Integer> terms ( )  
    //@ensures (* \result è un  
    //@generatore che dà gli  
    //@esponenti dei termini non  
    //@zero di this *);  
}  
  
}
```

```
public class IntSet {  
  //come visto prima, e in più:  
  public Iterator<Integer> elements( )  
  
    //@ensures (* \result è un  
    //@generatore che  
    //@dà i valori contenuti in this,  
    //@ciascuno 1 sola volta  
  
    //@requires (* this non deve  
    //@essere modificato quando  
    //@generatore è in uso *);
```

Implementazione con classe interna

```
public class Poly {  
    private int [ ] trms;  
    private int deg;  
    public Iterator<Integer> terms() {  
        return new PolyGen(this);  
    }  
    //classe interna  
    private static class PolyGen implements Iterator<Integer> {  
        ...  
    }  
    ...  
    //tutte le altre operazioni della classe  
}
```


static: PolyGen associata alla classe Poly.
(altrimenti ogni oggetto PolyGen associato a un oggetto Poly)
Non necessario

non posso generare esemplari fuori dalla classe Poly, ma PolyGen può accedere ai private di un Poly

Implementa interfaccia Iterator: ha next(), hasNext()

La classe interna per l'iteratore

```
private static class PolyGen implements Iterator<Integer> {  
    private Poly p; //il Poly da iterare  
    private int n; //prox el. da considerare  
    PolyGen(Poly lui) {  
        //@requires lui!=null;  
        p=lui;  
        if(p.trms [0]==0)n=1; else n=0;  
        //n=1 serve per hasNext() se p.deg ==0  
    }  
    public boolean hasNext ( ) {return n<=p.deg;}  
    public int next ( ) throws NoSuchElementException {  
        for (int e=n; e<=p.deg; e++)  
            if (p.trms[e] != 0) {n=e+1; return new Integer(e);}  
        throw new NoSuchElementException("Poly.terms");  
    }  
}
```



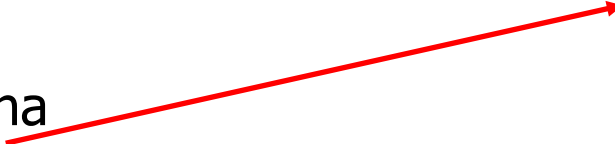
**Implementa
interfaccia
Iterator: ha next(),
hasNext()**

Iteratori "stand alone"

- Un iteratore potrebbe NON far parte di una classe, ma essere una procedura statica "stand alone"
- Per esempio, iteratore allFibos che genera tutti i numeri di Fibonacci
 - o *generatore* di numeri primi del testo

```
public class Num {  
    public static Iterator allFibos( ) {  
        return new FibosGen( );  
    }  
}
```

static necessario:
costruttore chiamato da
un metodo statico!



//classe interna

```
private static class FibosGen implements Iterator {  
    private int prev1, prev2; //i due ultimi generati  
    private int nextFib; //nuovo # Fibonacci generato
```

```
    FibosGen( ) {prev2=1; prev1=0;}
```

```
    public boolean hasNext ( ) {return true;}
```

```
    public Object next ( ) {  
        nextFib=prev1+prev2;  
        prev2=prev1; prev1=nextFib;  
        return Integer(nextFib);  
    }
```

NB: versione "raw"
del tipo Iterator



```
    }  
}
```

Uso dell'iteratore per i # Fibonacci

```
//@ensures (*stampa tutti # Fibonacci <=m in ordine
//@ crescente *);
public static void printFibos(int m); {
    Iterator g = Num.AllFibos();
    while (g.hasNext()) { //sempre true...
        int p = ((Integer) g.next()).IntValue();
        if (p > m ) return;
        System.out.println("prox Fibonacci e` “ + p);
    }
}
```


RI e AF per iteratori

- Simili a quelli di un ADT ordinario
- Caso di PolyGen
 - $RI(c) \rightarrow c.p \neq \text{null} \ \&\& \ (0 \leq c.n \leq c.p.\text{deg}+1)$
 - Oppure in JML:
 - `//@ private invariant c.p != null &&`
 - `//@ 0 <= c.n && c.n <= c.p.deg+1;`
 - espresso in termini delle instance variable di Poly
 - AF: la sequenza degli elementi ancora da generare
 - $\rightarrow AF(c)=[x_1, \dots, x_k]$, sequenza che contiene tutti e soli gli elementi $\neq 0$ che in $c.p.\text{trms}$ occupano posizione $j \geq c.n$, nello stesso ordine

Osservazioni

- “An iterator is an object whose job is to move through a sequence of objects and select each object in that sequence without the client programmer knowing or caring about the underlying structure of that sequence. In addition, an iterator is usually what’s called a “light-weight” object: one that’s cheap to create”
- Collezione può avere più tipi di iteratori (e quindi più metodi per generarli)
- Esempio: una collezione potrebbe avere:

```
//@ensures (* \result è generatore dall'elemento più piccolo al più grande *);  
public Iterator smallToBig() {}
```

```
//@ ensures (* \result generatore dall'elemento più grande *);  
public Iterator bigToSmall() {}
```

```
//@ensures (* \result è generatore dal primo elemento  
//@ nell'ordine di inserimento *);  
public Iterator first() {
```

...

Collezioni mutabili e metodo remove

- `remove` è un'operazione “opzionale”: se non è implementata, quando invocata viene lanciata **UnsupportedOperationException**
- In effetti, raramente è utile modificare una collezione durante un'iterazione: più spesso si modifica alla fine (e.g., trovo elemento e lo elimino, poi iteratore non più usato)
- Semantica di `remove()` assicurata dal “contratto” dell'interfaccia:
 - elimina dalla collezione l'ultimo elemento restituito da `next()`, a patto che
 - venga chiamata una sola volta per ogni chiamata di `next()`
 - collezione non venga modificata in qualsiasi altro modo (diverso dalla `remove`) durante l'iterazione
 - altrimenti
 - lanciata eccezione **IllegalStateException** se metodo `next()` mai chiamato o se `remove()` già invocata dopo ultima chiamata di `next()`
 - semantica della `remove()` non specificata se collezione modificata in altro modo (diverso dalla `remove`) durante l'iterazione
 - vincolo ragionevole: è prevedibile che una modifica durante un'iterazione lasci la collezione in uno stato inconsistente

Interfaccia Iterable (1)

- Per standardizzare l'uso degli iteratori, in Java esiste anche l'interfaccia Iterable (in java.lang)

```
public interface Iterable <T> {  
    // @ ensures (* restituisce un iteratore su una collezione di  
    //     elementi di tipo T *);  
    public Iterator <T> iterator ();  
}
```

- Le classi che implementano l'interfaccia Iterable sono tipicamente collezioni che forniscono un metodo di nome standard iterator() per ottenere un oggetto iteratore, con cui scandirne gli elementi

Interfaccia Iterable (2)

- Un contenitore che implementa l'interfaccia Iterable può essere argomento di un'istruzione for-each
 - il codice

```
for ( Tipo  variabile :  oggettoContenitore )  
    corpoDelCiclo
```
 - viene tradotto dal compilatore nel seguente

```
Iterator <Tipo> iter = oggettoContenitore.iterator();  
while ( iter.hasNext() ) {  
    Tipo variabile = iter.next();  
    corpoDelCiclo;  
}
```
- E' necessario implementare Iterable se si vuole costruire una collezione da usare con for-each
 - è un'utile semplificazione per gli *utenti* della collezione: per iterare su di essa, se non devono fare remove e non vogliono seguire un ordine particolare, non devono generare esplicitamente un iteratore, perchè usando un ciclo for-each lo fanno fare al compilatore
 - il prezzo della semplificazione lo paga l'implementatore, che deve codificare il metodo iterator()