Astrazione sul controllo: gli iteratori

Gli iteratori

- ✓ perché vogliamo iterarare “in modo astratto”
- ✓ iteratori e generatori in Java
  - specifica
  - utilizzazione
  - implementazione
  - rep invariant e funzione di astrazione
  - un esempio

Perché vogliamo iterarare “in modo astratto”

- ✓ problema: iterare su tipi di dato arbitrari
- ✓ esempio: calcolare la somma di tutti gli elementi di un `IntSet`

```
public static int setSum (IntSet s) throws
   NullPointerException
   // EFFECTS: se s è null solleva
   // NullPointerException altrimenti
   // ritorna la somma degli elementi di s
```

Soluzione insoddisfacente 1

```
public static int setSum (IntSet s) throws
   NullPointerException {
  // EFFECTS: se s è null solleva
  // NullPointerException altrimenti
  // ritorna la somma degli elementi di s
   int[ ] a = new int [s.size( )];
   int sum = 0;
   for (int i = 0; i < a.length; i++)
    {a[i] = s.choose( );
     sum = sum + a[i];
     s.remove(a[i]); }
  // risistema s
   for (int i = 0; i < a.length; i++)
    s.insert(a[i]);
   return sum;}
```

- ad ogni iterazione vengono chiamate due operazioni (`choose` e `remove`)
- gli elementi rimossi vanno reinseriti

Soluzione insoddisfacente 2

- potremmo realizzare `setSum` come metodo della classe `IntSet`
  - in modo più efficiente
    - accedendo la rappresentazione
  - non è direttamente collegata al concetto di `IntSet`
  - quante altre operazioni simili dovremmo mettere in `IntSet`?
    - trovare il massimo elemento.....

Soluzione insoddisfacente 3

```
public int [ ] members ()
 // EFFECTS: restituisce un array contenente gli
 // elementi di this, ciascuno esattamente una volta,
 // in un ordine arbitrario
public static int setSum (IntSet s) {
   int[ ] a = s.members();
   int sum = 0;
   for (int i = 0; i < a.length; i++) sum = sum + a[i];
   return sum;}
```

✓ inefficiente
- due strutture dati
- non sempre vogliamo generare tutti gli elementi della collezione
  - c'è un elemento maggiore di x?

Altre soluzioni insoddisfacenti

- ✓ dotiamo `IntSet` di una operazione che ritorna la rappresentazione
  - distruggiamo l'astrazione
- ✓ ridefiniamo l'astrazione `IntSet` in modo da avere una nozione di indiciamento
  - è un'astrazione molto più complessa e non direttamente legata alla nozione di insieme

Di cosa abbiamo bisogno?

- ✓ un meccanismo generale di iterazione
  - facile da usare
  - efficiente
  - che preservi l'astrazione
- ✓ per ogni i prodotto da g esegui a su i
- ✓ g è un *generatore* che produce in modo incrementale (uno alla volta) tutti gli elementi i della collezione corrispondente all'oggetto
- ✓ l'azione a da compiere sugli elementi è separata dalla generazione degli elementi stessi

Iteratori e ordine superiore

- per ogni i prodotto da g esegui a su i
- cose di questo genere si realizzano molto facilmente con la normale astrazione procedurale in quei linguaggi (tipicamente funzionali) in cui 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: mapcar in LISP

Iteratori in Java

- per ogni i prodotto da g esegui a su i
- i generatori sono oggetti di tipo `Iterator`
  - il tipo `Iterator` è definito dalla seguente interfaccia Java (`java.utilpackage`)

```
public interface Iterator {
   public boolean hasNext ( );
// EFFECTS: restituisce true se ci sono altri elementi
// altrimenti false
   public Object next throws NoSuchElementException;
// EFFECTS: se ci sono altri elementi da generare dà il
// successivo e modifica lo stato di this, altrimenti
// solleva NoSuchElementException (unchecked)
}
```

  - possiamo definire metodi che restituiscono generatori

Come si usano i generatori 1

- per ogni i prodotto da g esegui a su i
- il metodo `primesLT100`

```
public static Iterator primesLT100 ()
// EFFECTS: restituisce un generatore,
// che genera incrementalmente tutti i
// numeri primi (Integer) minori di 100
```

- può essere utilizzato per realizzare un'iterazione astratta

```
// ciclo controllato da hasnext
Iterator g = primesLT100 ();
while (g.hasNext())
   {int x = ((Integer) g.next( )).intValue( );
   // usa  x }
```

Come si usano i generatori 2

```
public static Iterator primesLT100 ()
// EFFECTS: restituisce un generatore,
// che genera incrementalmente tutti i
// numeri primi (Integer) minori di 100
```

- ✓ può essere utilizzato per realizzare un'iterazione astratta

```
// ciclo controllato da  exception
Iterator g = primesLT100();
try {while (true) {int x =
     ((Integer) g.next()).intValue();
     // uso di x
     }
catch (NoSuchElementException e) { };
```

Specifica dei metodi che restituiscono generatori

- ✓ spesso chiamati *iteratori*
  - da non confondere con il tipo `Iterator` che restituiscono
- ✓ possono essere procedure stand alone
  - come `primesLT100`
- ✓ più interessante quando sono metodi di una classe che definisce una astrazione sui dati
  - vediamo degli esempi su `IntSet` e `Poly`

Specifica di un iteratore per `Poly`

```
public class Poly {
// come prima più
 public Iterator terms ()
 // EFFECTS: ritorna un generatore che produrrà gli
 // esponenti dei termini diversi da 0 in this (come
 // Integers) fino al grado del polinomio, in ordine
 // crescente
                                    }
```

✓ un tipo di dato può avere anche più iteratori

Specifica di un iteratore per `IntSet`

```
public class IntSet {
// come prima più
 public Iterator elements ()
  // EFFECTS: ritorna un generatore che produrrà tutti
  // gli elementi di this (come Integers) ciascuno una
  // sola volta, in ordine arbitrario
  // REQUIRES: this non deve essere modificato
  // finché il generatore è in uso
}
```

- ✓ la clausola REQUIRES impone condizioni sul codice che utilizza il generatore
  - per questo è messa alla fine
  - tipica degli iteratori su tipi di dato modificabili

Specifica di un iteratore stand alone

```
public class Num {
// come prima più
 public static Iterator allPrimes ()
 // EFFECTS: ritorna un generatore che produrrà tutti
 // i numeri primi (come Integers) ciascuno una
 // sola volta, in ordine arbitrario
                        }
```

- ✓ il limite al numero di iterazioni deve essere imposto dall'esterno
  - il generatore può produrre infiniti elementi

Utilizzazione degli iteratori 1

```
public Iterator terms ()
 // EFFECTS: ritorna un generatore che produrrà gli
 // esponenti dei termini diversi da 0 in this (come
 // Integers) fino al grado del polinomio, in ordine
 // crescente
public class Comp {
 public static Poly diff (Poly p) throws NullPointerException
  // EFFECTS: se p è null solleva NullPointerException
  // altrimenti ritorna il poly ottenuto differenziando
  // p
   {Poly q = new Poly();
   Iterator g = p.terms();
   while (g.hasNext())  {
   int exp = ((Integer) g.next()).intValue();
   if (exp == 0) continue; // ignora il termine 0
   q = q.add (new Poly(exp*p.coeff(exp), exp-1));
   return q;}}
```

✓ implementazione di `diff` esterna alla classe `Poly`

Utilizzazione degli iteratori 2

```
public static Iterator allPrimes ()
 // EFFECTS: ritorna un generatore che produrrà tutti
 // i numeri primi (come Integers) ciascuno una
 // sola volta, in ordine arbitrario

public static void printPrimes (int m) {
 // MODIFIES: System.out
 // EFFECTS: stampa tutti i numeri primi minori o uguali a m
 // su System. out
   Iterator g = Num.allPrimes();
   while (true) {
   Integer p = (Integer) g.next();
   if (p.intValue() > m) return; // forza la terminazione
   System.out.println("The next prime is: " + p.toString()); }
   }
```

Utilizzazione dei generatori

```
public static int max (Iterator g) throws EmptyException,
   NullPointerException {
// REQUIRES: il generatore g genera (contiene) solo Integers
// EFFECTS: se g è null solleva NullPointerException; se g è
// vuoto solleva EmptyException, altrimenti consuma tutti gli
// elementi di g e restituisce il massimo intero in g
try {int m = (Integer) g.next()).intValue();
    while (g.hasNext( ))
     {int x = (Integer) g.next()).intValue();
      if (m < x) m = x; } return m;}
catch (NoSuchElementException e)
{throw new EmptyException("Comp.max"); }  }
```

- ✓ i generatori (essendo oggetti) possono essere passati come argomento a metodi che cosí astraggono da dove provengono gli argomenti su cui lavorano
  - prodotti da `elements` di `IntSet`, `primesLT100`, ...

Implementazione degli iteratori e dei generatori

- i generatori sono oggetti che hanno come tipo un sottotipo di `Iterator`
  - istanze di una classe γ che "implementa" l'interfaccia `Iterator`
- un iteratore α è un metodo (stand alone o associato ad un tipo astratto) che ritorna il generatore istanza di γ
  - γ deve essere contenuta nello stesso modulo che contiene α
    - dall'esterno del modulo si deve poter vedere solo l'iteratore α (con la sua specifica)
    - non la classe γ che definisce il generatore
- γ deve avere una visibilità limitata al package che contiene α
  - oppure può essere contenuta nella classe che contiene α
    - come inner class privata
- dall'esterno i generatori sono visti come oggetti di tipo `Iterator`
  - perché il sottotipo γ non è visibile

Classi nidificate

- una classe γ dichiarata 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 le variabili di istanza ed i metodi di istanza di α
  - le classi che definiscono i generatori si possono quasi sempre definire come inner classes statiche

Classi nidificate: semantica

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

Implementazione degli iteratori 1

```
public class Poly {
 private int[ ] termini;
 private int deg;
 public Iterator terms ()  {return new PolyGen(this); }
 // EFFECTS: ritorna un generatore che produrrà gli
 // esponenti dei termini diversi da 0 in this (come
 // Integers) fino al grado del polinomio, in ordine  crescente
 private static class PolyGen implements Iterator {
  // inner class (classe annidata) statica (l'array va passato al costruttore)
   private Poly p; // il Poly su cui si itera
   private int n; // il prossimo termine da considerare
   PolyGen (Poly it) {
   // REQUIRES: it != null
     p = it; if (p.termini[0] == 0)  n = 1; else n = 0; }
   public boolean hasNext ()  {return n <= p.deg; }
   public Object next () throws NoSuchElementException {
   for (int e = n; e <= p.deg; e++)
    if (p.termini[e] != 0){n = e + 1; return new Integer(e); }
   throw new NoSuchElementException("Poly.terms"); }  }  }
```

Implementazione degli iteratori 2

```
public class Poly {
 private int[ ] termini;
 private int deg;
 public Iterator terms ()  {return new PolyGen(); }
 // EFFECTS: ritorna un generatore che produrrà gli
 // esponenti dei termini diversi da 0 in this (come
 // Integers) fino al grado del polinomio, in ordine  crescente
 private class PolyGen implements Iterator {
  // inner class (classe annidata) di istanza (vede termini)
   private int n; // il prossimo termine da considerare
   PolyGen () {
   // REQUIRES: it != null
     if (termini[0] == 0)  n = 1; else n = 0; }
   public boolean hasNext ()  {return n <= deg; }
   public Object next () throws NoSuchElementException {
    for (int e = n; e <= deg; e++)
     if (termini[e] != 0) {n = e + 1; return new Integer(e); }
    throw new NoSuchElementException("Poly.terms"); }  }  }
```

Implementazione degli iteratori 3

```
public class Num {
 public static Iterator allPrimes (){return new PrimesGen();}
 // EFFECTS: ritorna un generatore che produrrà tutti
 // i numeri primi (come Integers) ciascuno una
 // sola volta, in ordine arbitrario
 private static class PrimeGen implements Iterator {
  // inner class (classe annidata)
    private Vector ps; // primi già dati
    private int p; // prossimo candidato alla generazione
    PrimesGen () {p = 2; ps = new Vector(); }
    public boolean hasNext ()  {return true }
    public Object next () {
     if (p == 2) { p = 3; return new Integer(2);}
     for (int n = p; true; n = n + 2)
       for (int i = 0; i < ps.size(); i++){
         int e1 = ((Integer) ps.get(i)).intValue();
         if (n%e1 == 0) break; // non è primo
         if (e1*e1 > n)
          {ps.add(new Integer(n); p = n + 2;
          return new Integer(n);}}}  }}
```

Classi nidificate e generatori

- ✓ le classi i cui oggetti sono generatori definiscono comunque dei tipi astratti
  - sottotipi di `Iterator`
- ✓ in quanto tali dovrebbero essere dotati di
  - una funzione di astrazione
  - un invariante di rappresentazione

Funzione di astrazione per i generatori

- ✓ dobbiamo sapere cosa sono gli stati astratti
- ✓ per tutti i generatori, lo stato astratto è
  - la sequenza di elementi che devono ancora essere generati
  - la funzione di astrazione mappa la rappresentazione su tale sequenza

Funzione di astrazione 1

```
public class Poly {
 private int[ ] termini;
 private int deg;
 public Iterator terms ()  {return new PolyGen(this); }
 private static class PolyGen implements Iterator {
  // inner class (classe annidata)
   private Poly p; // il Poly su cui si itera
   private int n; // il prossimo termine da considerare
 // la funzione di astrazione
 // α(c) = [ x1, x2, ...] tale che
 // ogni xi è un Integer, e
 // gli xi sono tutti e soli gli indici i>=n
 // per cui c.p.termini[i] != 0, e
 // xi > xj per tutti gli i > j >= 1
```

✓ notare che si usano le rep sia di `Poly` che di `Polygen`

Invariante di rappresentazione 1

```
public class Poly {
 private int[ ] termini;
 private int deg;
 public Iterator terms ()  {return new PolyGen(this); }
 private static class PolyGen implements Iterator {
  // inner class (classe annidata)
   private Poly p; // il Poly su cui si itera
   private int n; // il prossimo termine da considerare
 // l'invariante di rappresentazione:
 // I(c) = c.p != null e
 // (0 <= c.n <= c.p.deg)
```

✓ notare che si usano le rep sia di `Poly` che di `Polygen`

Funzione di astrazione 2

```
public class Num {
 public static Iterator allPrimes (){return new PrimesGen();}
 private static class PrimeGen implements Iterator {
  // inner class (classe annidata)
   private Vector ps; // primi già dati
   private int p; // prossimo candidato alla generazione
 // la funzione di astrazione
 // α(c) = [ p1, p2, ...] tale che
 // ogni pi è un Integer, è primo, ed è >= c.p e
 // tutti i numeri primi >= c.p occorrono nella
 //       sequenza, e
 // pi > pj per tutti gli i > j >= 1
```

Invariante di rappresentazione 2

```
public class Num {
 public static Iterator allPrimes (){return new PrimesGen();}
 private static class PrimeGen implements Iterator {
  // inner class (classe annidata)
   private Vector ps; // primi già dati
   private int p; // prossimo candidato alla generazione
 // l'invariante di rappresentazione:
 // I(c) = c.ps != null e
 // tutti gli elementi di c.ps sono primi e
 //      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 dato astratto è modificabile ci dovrebbe sempre essere il vincolo sulla non modificabilità del dato 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 tasks in attesa su una coda di tasks

```
Iterator g = q.allTasks();
while (g.hasNext()) {
     Task t = (Task) 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

Esercizio per casa

- ✓ implementazione di generatore e iteratore per `IntSet`
  - inclusi invariante e funzione di astrazione