Iteration Abstraction

Comp-303 : Programming Techniques

Lecture 8

Alexandre Denault
Computer Science
McGill University

Winter 2004

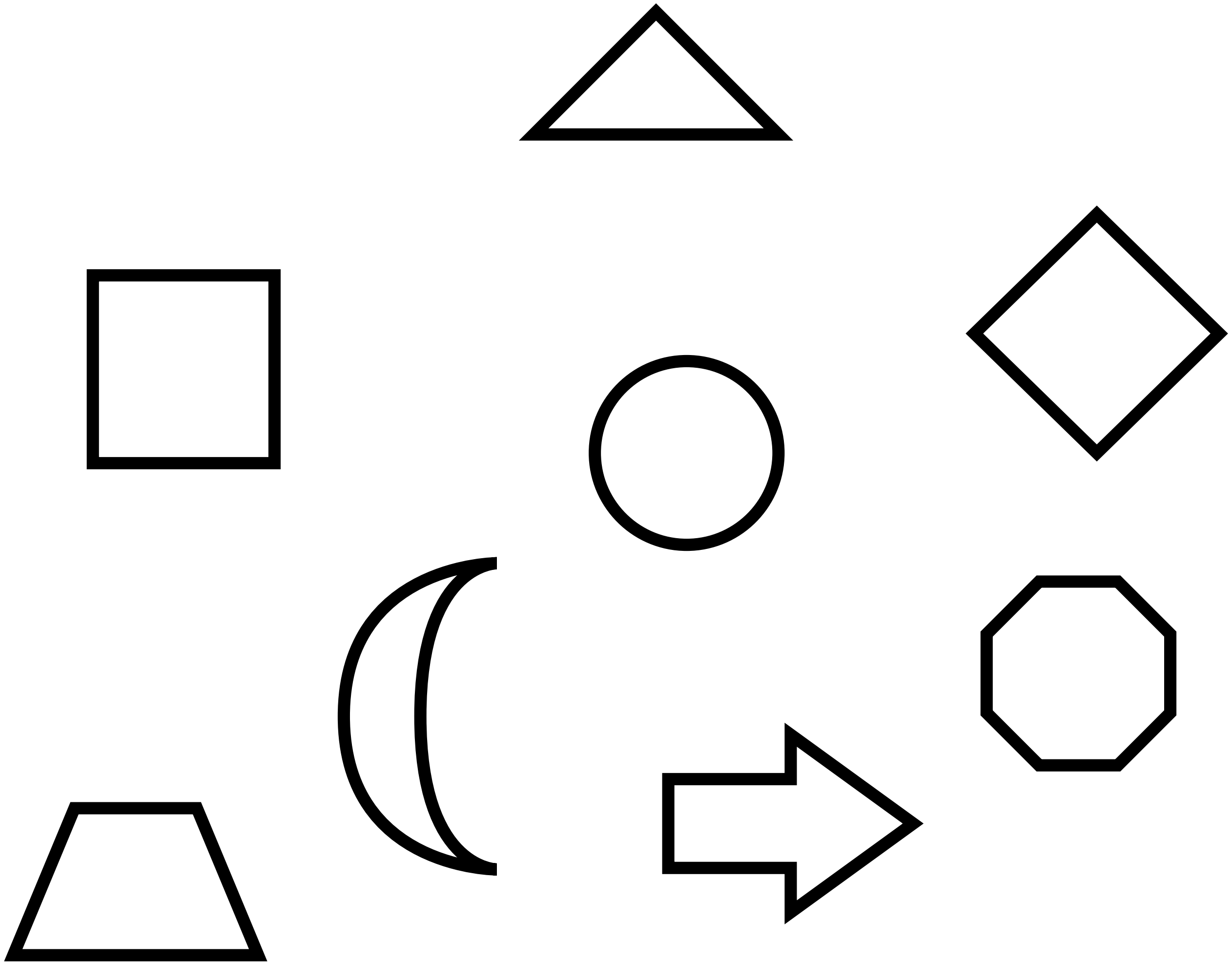
Last lecture . . .

- Exceptions are thrown under exceptional conditions. The should not be used for regular control flow.
- Exceptions move conditions from REQUIRES clause to EFFECTS clause.
- Checked exceptions are declared in method header and must be caught or propagated by caller.
- Unchecked exceptions do not have to be declared and are propagated automatically.
- A caught exception can be reflected or masked.
- When using Defensive Programming, all sources of errors must be checked, even the unlikely and the impossible.

Announcements . . .

- The project Req&Spec Document and the first Assignment are due February 3rd.
  - Req&Spec Document before end of class.
  - Assignment 1 paper in drop off box before 23:55 (or before the T.A. picks it up Wednesday)
  - Assignment 1 electronic before 23:55 on Web CT (or you will be using your late days)
- Massive update of the website.
- At then end of the lecture, I'll talk a little about February.

The Iteration Game



Iteration Abstraction

- It is often necessary to perform some action on all elements of a collection

```
for all elements of an IntSet
   do action
```

- For example, the following function will need to visit all the elements of an IntSet.

```
public static int getTotal (IntSet s)
   throws NullPointerException
    // EFFECT: If s is null throws
    // NullPointerException else returns
    // the sum of the elements of s
```

A first implementation of setSum

```
public static int getTotal (IntSet s)
    throws NullPointerException {

    int [ ] a = new int[s.size()];
    int sum = 0;

    // save each element from the set into a, sum it, remove it
    for (int i = 0; i < a.length; i++) {
       a[ i ] = s.choose ( );
       sum = sum + a [ i ];
       s.remove(a [ i ]);
    }

    // restore elements of s
    for (int i = 0; i < a.length; i++)
       s.insert( a [ i ]);

    return sum;
}
```

Problems with this approach

- We need 3 calls per element: choose + remove + insert.
- This is inefficient.
- We could implement getTotal within the IntSet datatype.
- However,
  - This is not a general operation.
  - It is impossible to foresee all ways to manipulate all elements of IntSet.
  - There might be more than one way to calculate the sum.
- Instead, we could create an operation returning array of elements.

Second implementation of getTotal

```
public static int getTotal (IntSet s) {
    // a = array with all elements of s

    int [ ] a = s.members ( );
    int sum = 0;

    // sum all elements of array
    for (int i = 0; i < a.length; i++) {
        sum = sum + a [ i ];
    }

    return sum;
}
```

Problems with this approach

- This is inefficient if IntSet is large.
- It requires the creation of a large data structure just to iterate over all elements.
- Sometimes not all elements are needed. Too much work is done in advance. e.g. sum all the negative elements
- Instead, we could return the representing vector of IntSet.
- This exposes the representation and opens up all kinds of abuse by user program. In other words, it breaks the abstraction.

Problems with all approaches

- None of these solutions are general.
- Iterating over a set of Integers should be similar to iterating over
  - a bag of Integers
  - an array of Integers
  - a vector of Integers
- In our case, iteration could be described as

  > Give me each Integer in this collection, one by one, in some order.

- This is iteration abstraction.

An iterator object, as defined by Java

- Iterators are defined in Java.Util
- The NoSuchElementException is an unchecked exception.

```
public interface Iterator {
   public boolean hasNext ( );
      // EFFECTS: Returns true if there are no more
      // elements else returns false

   public Object next ( );
      throws NoSuchElementException;
      // MODIFIES: this
      // EFFECTS: If there are more results to yield, returns
      // the next result and modifies the state of this to
      // to record the yield.
      // Otherwise, throws NoSuchElementException
}
```

IntSet with iterator specification

```
public class IntSet {
   // as before plus:

    public Iterator elements ( )
        // EFFECT: Returns a generator that will produce
        // all elements of this (as Integers), each exactly
        // once, in arbitrary order.
        // REQUIRES: this must not be modified while the
        // generator is in use.
}
```

Iterators and generators

- An *iterator* is a procedure that returns a generator object. A data abstraction can have one or more iterator methods.
- A *generator* (implements java.util.Iterator) is an object that produces the elements. It has methods to get the next element and to determine whether there are any more elements. The generator's type is a subtype of Iterator. As an object it can be passed to other methods.
- The specification of an iterator defines the behavior of the generator; a generator has no specification of its own. The iterator specification often includes a requires clause at the end constraining the code that uses the generator.

Third implementation of getTotal

```
public static int getTotal (IntSet s) {

   Iterator g = s.elements ( );
   int sum = 0;

   while (g.hasNext( ))
      sum = sum + ((Integer) g.next( )) . intValue;

   return sum;
}
```

Third implementation of getTotal / exceptions

```
public static int getTotal (IntSet s) {

   Iterator g = s.elements ( );
   int sum = 0;

   try {
      while (true) sum = sum + ((Integer) g.next( )) . intValue;
   } catch (NoSuchElementException e) { }

   return sum;
}
```

Implementing Iterators

- An Iterator's implementation requires a class for the associated generator.
- The generator class is a inner class: it is nested inside the class containing the iterator and can access the private information of its containing class (when that information is passed through the iterator procedure).
- The generator class defines a subtype of (implements) the Iterator interface.
- The implementation of the generator assumes that using code obeys constraints imposed on it by the requires clause of the iterator.

Implementation of elements Iterator

```
private Vector els;

public Iterator elements ( )
   { return new IntGenerator (this); }

// inner class
private class IntGenerator implements Iterator {

   private IntSet s;  // the IntSet being iterated
   private int n;  // index of the next element to consider

   IntGenerator (IntSet is) {
       // REQUIRES: is != null

       s = is;
       n = 0;
   }

   public boolean hasNext ( ) { return n < s.els.size(); }
```

Implementation of elements Iterator

```
    public Object next ( ) throws NoSuchElementException {

       if ( n < s.els.size() ) {
          Integer result = s.els.get( n );
          n++;
          return result;
       } else {
           throw NoSuchElementException("IntSet.elements");
       }
    }
 }
```

Implementation of terms Iterator

```
private int[ ] trms;
private int deg;

public Iterator terms ( ) { return new PolyGen (this); }

    // inner class
    private class PolyGen implements Iterator {

       private Poly p;  // the IntSet being iterated
       private int n;  // index of the next term to consider

       PolyGen (Poly it) {
          // REQUIRES: it != null

          p = it;
          if (p.trms[0] ==0) n = 1; else n = 0;
       }
```

Implementation of terms Iterator

```
  public boolean hasNext ( ) { return n <= p.deg; }

  public Object next ( ) throws NoSuchElementException {

      for ( int e = n; e <= p.deg; e++ )
      if (p.trms[e] != 0) {
          n = e + 1;
          return new Integer (e);
      }
      throw NoSuchElementException("Poly.terms");
   }
}
```

An iterator to generate Prime numbers

```
public class Num {
   public static Iterator allPrimes ( )
      { return new PrimesGenerator ( ); }

   // inner class
   private class PrimesGenerator implements Iterator {
      private Vector ps;  // primes yielded
      private int p;  // next candidate

      PrimesGenerator ( ) { p = 2; ps = new Vector ( ); }

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

An iterator to generate Prime numbers

```
      public Object next ( ) {
         if (p == 2) { p = 3; return 2; }

         for (int n = p; true ; n = n + 2) {
             for (int i = 0; i < ps.size ( ); i++) {
                int el = ((Integer) ps.get(i)).intValue( );
                if (n % el) == 0) break; // not a prime
                if (el * el) > n) {
                   // n is a prime
                   ps. add(new Integer(n));
                   p = n + 2;
                   return n;
                }
             }
          }
       } // end of function
    } // end of inner class
} // end of class
```

Iterators over Iterators

```
13
14
18
23
25
29
36
41
52
65
66
78
80
95
101
```

Iterators over Iterators

- Iterators can be extended by combining them with other iterators.

```
static Iterator filter (Iterator g, int x)
   throws nullPointerException
   // REQUIRES: g contains only Integers
   // MODIFIES: g
   // EFFECTS: if g is null throws nullPointerException
   // else returns a generator that produces each
   // element e of g for e/x = 0
```

- Now it is easy to iterate the divisors of x using elements of an IntSet that contains natural numbers.

Functions are objects

- An iterator is a special case of a general principle: it can be useful to treat an operation as an object that can be passed around just like any other object.

```
static Iterator dynamicfilter (Iterator g, Check x)
   throws nullPointerException
   // REQUIRES: g contains only Integers
   // MODIFIES: g
   // EFFECTS: if g is null throws nullPointerException
   // else returns a generator that produces each
   // element e of g for which x.checker(e) is true

interface Check {public boolean checker  ( Integer i);}
```

- Now it is easy to iterate over primes numbers using elements of an IntSet that contains natural numbers.

Design issues

- Most data types that store a collection of items will include iterators.
- Adequacy requires that elements can be accessed efficiently and conveniently
- Mutable collections require that the loop using a generator does not change the collection
- However, the standard Iterator interface allows a modification operation: *void remove()*

Optional operation remove()

- The method *public 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.
- This method can throw:
  - *UnsupportedOperationException* : if the remove operation is not supported by this Iterator.
  - *IllegalStateException* : if the next method has not yet been called, or the remove method has already been called after the last call to the next method.

Useful modifications (this should be exceptional)

```
Iterator g = q.allTasks( );
while (g.hasNext( )) {
Task t = (Task) g.next ( );
// perform t
// if t generates a new task nt
// enqueue it by performing q.enq(nt)
}
```

- The generator should be implemented so that it is aware of the new task.

Summary

- Adequacy of collection types requires a way to iterate efficiently and conveniently over its elements.
- Iterators provide a that solution.
- A generator object returns elements from the collection one at a time, usually without requiring extra storage or requiring access to all elements.
- Iterators support abstraction by hiding how elements are produced: the generator has access to private variables of the collection but shields the user from this knowledge
- Iterators assume that the collection remains unchanged while iterating, except through the optional remove() operation

Something new for February

Topics of interest

- Network / Socket programming
- Game design (interface)
- Obfuscation
- Threading
- Reflection
- Serialization

Tool of the day : Thinkfree Office

- The award-winning ThinkFree Office is an affordable suite of word processing, spreadsheet, and presentation graphics applications.
- It can open, edit, and save directly to the corresponding Microsoft(R) Office file formats like .doc, .xls, and .ppt.
- Its unique, pure Java architecture enables it to run on Windows, Linux or Macintosh operating systems.
- ThinkFree Office features integrated, Internet-based file sharing and storage with end-to-end security
- More information on Thinkfree Office can be found at

  `http://www.thinkfree.com/`