Lecture 09 Iterators and Enumerations

Reading for these lectures:
Weiss, Section 6.3.2 Iterator Interface.

Much better is: ProgramLive, Section 12.3.

The introduction of suitable abstractions is our only mental aid to reduce the appeal to enumeration, to organize and master complexity. *Edsger W. Dijkstra*

Incrementing C by 1 is not enough to make a good object-oriented language. *M. Sakkinen*

(in C, C++, and Java, x++ adds 1 to x.)

Practice practice practice

It will help you tremendously to download the programs that we look at today and experiment with them.

It will help to write a few implementations of class Iterator yourself and test them. Examples:

An Iterator of the digits in a String, producing objects of class Integer.

An Iterator of the prime numbers.

An Iterator of the chars of a String from end to beginning.

An Iterator for the Fibonacci numbers.

Abstract data type

Type: a bunch of values together with operations on them.

We can write an interface that DEFINES such a type. We call it an “abstract data type” because we don’t give a concrete implementation of it, we just define it “abstractly”

Type List211

```
/** A list is a sequence of Objects. */
public interface List211 {
   /** Make the list empty */
   void makeEmpty();

   /** Add item e to the list */
   void add(Object e);

   /** Delete an item from the list */
   void delete();

   /** = an item in the list  */
   Object getItem();

   /** = the number of items in the list */
   int size();
}
```

Operations are not only abstract but ambiguous. Doesn’t say which item to delete or where to add an item.

Implement list as a stack

```
/** An instance is a stack s of Objects */
public class Stack2 implements List211 {
   private int N; // Maximum number of values in s
   private Object[] b; // Stack s is in b[0..n-1]
   private int n;

/** Constructor: empty stack with at most N elements */
   public Stack2(int N) {  }

   /** Make the stack empty */
   public void makeEmpty() {  }

   /** add item e to the front of the stack */
   public void add(Object e) {  }

   /** delete item from the front of the stack */
   public void delete() { }

   /** = the front item of the stack  */
   public Object getItem() {  return null;  }

   /** = the number of items in the stack */
   public int size() {  return 0;  }

    /** = stack elements, with top first, separated by
            “,” and delimited by “[“ and “]”  */
   public String toString() {  return null;  }
}
```

This class is on the website. Download and play with it.

Class invariant: describes the fields and the values they contain:

```
/** An instance is a stack s of Objects, with a
     maximum size */
public class Stack1 implements List211 {
   /** N is the max size of stack s.
        Stack s appears in b[0..n-1], so
        that s contains n items. b[0] is the
        bottom and b[n-1] the top of the stack*/
   public Object[] b;
   public int n;

   /** Constructor: empty stack of <= m items */
   public Stack1(int m)  {
          n= m;
          b= new Object[m];
          n= 0;
   }
```

Interface Comparable

```
public interface Comparable {
    /** =  if this Object < ob  then a negative integer
            if this Object = ob, then 0
            if this Object > ob, then a positive integer */
    int compareTo (Object ob);
}
```

It has ONE method, compareTo.

Implementing method compareTo

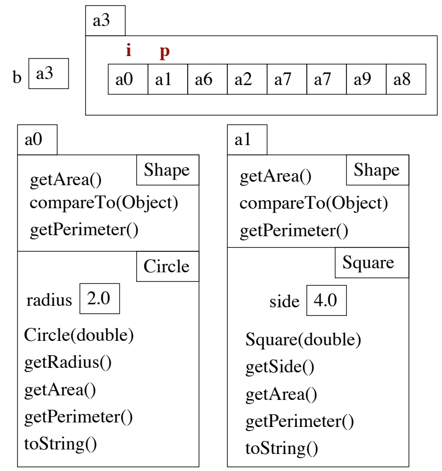
We look at a DrJava program that has abstract class Shape with two subclasses, Square and Circle. A main program creates an array of 8 shapes and sorts them. This illustrates nicely how the interface Comparable is used.

```
public class Shape implements Comparable {
  // =       if this Object < ob  then a negative integer
  //         if this Object = ob, then 0
  //         if this Object > ob, then a positive integer
  //        (it's expected that ob is really a Shape)
  int compareTo(Object ob) {
      if (this.getArea() < ((Shape)ob).getArea())
              return -1;
      if (this.getArea() ==((Shape)ob).getArea())
              return 0;
      return 1;
  }

   abstract int getArea();
}     (Shapes are ordered by their area)
```

ob is cast to Shape

Objects from the Shape Demo



b[i].compareTo(b[p])

b a3

a3: i p; a0 a1 a6 a2 a7 a7 a9 a8

a0: Shape: getArea() compareTo(Object) getPerimeter(); Circle: radius 2.0, Circle(double) getRadius() getArea() getPerimeter() toString()

a1: Shape: getArea() compareTo(Object) getPerimeter(); Square: side 4.0, Square(double) getSide() getArea() getPerimeter() toString()

Interface Iterator in package java.util.

(From dictionary)

Enumeration: an itemized list.

Enumerate: to relate one after another.

It is not necessary to *enumerate* all the bitter and factious disputes that marked this unhappy quarter century

He *enumerated* the advantages of his new position.

He *enumerated* the necessary qualities of a good general

A class that extends iterator is used to make it easy to write loops to enumerate a list of values.

Interface Iterator in package java.util.

```
public interface Iterator {
  /** = "there are more elements to enumerate */
  boolean hasNext();

  /** = the next element of the enumeration.
         throw NoSuchElementException if there
          are none */
  Object next();

  /** remove the last element returned */
  void remove();
}
```

```
/** An enumerator for the evens in h..k */
public class EvensEnum implements Iterator {
   private int item; /** next even int to enum. (if <= k)*/
   private int k;    /** enumerate the evens <= k

   /** Constructor: an enumerator for the evens in h..k  */
   public EvensEnum(int h, int k) {
      this.k= k;
      if (h % 2 == 0) item= h;
      else item= h+1;
   }

   /** = "there is another item to enumerate" */
   public boolean hasNext() { return item <= k; }

   /** = the next item to enumerate. Throw a
          NoSuchElementException if there is none.*/
   public Object next() {
      if (item > k)  throw new NoSuchElementException();
      int r= item;
      item= item + 2;
      return new Integer(r);
   }

   /** not implemented */
   public void remove() {}
}
```

Using the iterator

```
/** Print out the even integers in h..k */
public static void printEvens(int h, int k) {
   EnumerateEvens enum=
        new EnumerateEvens(h,k);
```

Create an instance of EnumerateEvens

```
   while (enum.hasNext()) {
        Object ob= enum.next();
        System.out.println(ob);
    }
}
```

Typical loop: each iteration gets a new item and processes it.

ALWAYS test whether there is another item before getting it.

Iterator to enumerate letters in a string

```
public class LetterEnum implements Iterator {

   private String s; /** enumerate letters of s*/
   private int k;    /** letters in s[k..] still to be enumerated.
             Calling hasNext sets it to index of next
             letter, if there is one, or to s.length().*/

   /** Constructor: an enumerator for letters in s  */
   public LetterEnum(String s)
     {  this.s= s;   k= 0; }

   /** = "there is another item to enumerate" */
   public boolean hasNext() {
      while (k != s.length() &&
            !Character.isLetter(s.charAt(k)))
          { k= k+1; }
      return k < s.length();
   }

   /** = the next item to enumerate. Throw a
       NoSuchElementException if there is none.*/
   public Object next() {
      if (!hasNext())
          return new NoSuchElementException();
      k= k+1;
      return new Character(s.charAt(k-1));
   }
```

Iterator to enumerate letters in a string

```
public class LetterEnum implements Iterator {

   private String s; /** enumerate letters of s*/
   private int k;    /** letters in s[k..] still to be enumerated.
             Calling hasNext sets it to index of next
             letter, if there is one, or to s.length().*/

   /** = "there is another item to enumerate" */
   public boolean hasNext() {
      while (k != s.length() &&
            !Character.isLetter(s.charAt(k)))
          { k= k+1; }
      return k < s.length();
   }
```

Calling hasNext has a benevolent side-effect: it sets k to index of next item to be enumerated. If k is already contains that index, then hasNext does not change k!!!

The last statement is important.

A method to produce a string that contains all the items enumerated by any Iterator!!!

```
 /** =  a string that contains the items enumerated
         by it, separated by "," and delimited by
         "[" and "]" */
public static String toString(Iterator it) {
       String res= "[";
       while (it.hasNext()) {
          if (res.length() > 1)
              res= res + ",";
          res= res + it.next();
       }
     return res + "]";
   }
```