Iterators & Comparators

Lecture 6

Menu

- Iterators and Iterables
- Sorting collections
- Comparators and Comparables

Iterators and Iterable

- The foreach loop requires an Iterable:

```
for (type var :  Iterable <type> ){
   … var …
}
```

eg, all Collections

```
Iterable <T>

  public Iterator<T> iterator();
```

```
Iterator <T>
        public boolean hasNext();
        public T next();
        public void remove();
```

```
Iterator<type > itr  = construct iterator

while (itr.hasNext() ){
   type var = itr.next();
   … var …
}
```

Creating Iterators

- Iterators are not just for Collection objects:
  - Anything that can generate a sequence of values
  - Scanner
  - Pseudo Random Number generator :

```
public class RandNumIter implements Iterator<Integer>{
      private int num  = 1,
      public boolean hasNext(){
          return true;
      }
      public Integer next(){
          num = (num * 92863) % 104729 + 1
          return num;
      }
      public void remove(){throw new
   UnsupportedOperationException();}
}
Iterator<Integer> lottery = new RandNumIter();
for (int i = 1; i<1000; i++)
      [illegible]
```

remove(): must be defined, but doesn't need to do anything!

Creating an Iterable

- An Iterable<T> is an object that provides an Iterator<T>:
  - eg: An ArithSequence representing an infinite arithmetic sequence of numbers, with a starting number and a step size,
    eg 5, 8, 11, 14, 17,….

```
public class ArithSequence implements Iterable<Integer>{
    private int start;
    private int step;
    public ArithSequence(int start, int step){
        this.start = start;
        this.step = step;
    }
    public Iterator<Integer> iterator(){
        return new ArithSequenceIterator(this);
    }
        :
```

Creating an Iterable

```
    private class ArithSequenceIterator implements Iterator<Integer>{
        private int nextNum;
        private ArithSequence source;
        public Ar                                 equence as){
            sourc
            nextN
        }
        public boolean hasNext(){
            return true;
        }
        public Integer next(){
            int ans = nextNum;
            nextNum += source.step;
            return ans;
        }
        public void remove(){throw new UnsupportedOperationException();}
    }   // end of ArithSequenceIterator class
}   // end of Arithmetic Sequence class
```

Class is only accessible from inside ArithSequence

Using the Iterable

- Can use the iterable object in the foreach loop:

```
for (int n : new ArithSequence(15, 8)){
    System.out.printf("next number is %d \n", n);
}
```

- Can use the iterator of the iterable object directly.

```
ArithSequence seq = new ArithSequence(15, 8));
Iterator<Integer> iter = seq.iterator();
processFirstPage(iter);
for (int p=2; p<maxPages; p++)
   processNextPage(p, iter);
```

Can pass iterator to different methods to deal with.

Working with Collections

- Done:
    - Declaring and Creating collections
    - Adding, removing, getting, setting, putting,….
    - Iterating through collections
        - [Iterators, Iterable, and the foreach loop]
- What else?
    - Sorting Collections
    - Implementing Collection classes

Sorting a collection

- What kinds of collections could you sort?
  - Set ?
  - Stack ?
  - Queue ?
  - List ?
  - Map ?

- How can you sort them?

Sorting in “Natural order”

- But what order will it sort into ?
  - “natural order of the values”
- Fine for Strings, Integer, Double
  - Strings ordered alphabetically, as in a phonebook (actually a little more complicated….)
  - Integer, Double ordered by numerical value
- But what’s the “natural order” of Faces in a crowd?
  - Answer:
    - Whatever you defined it to be, if you defined it.
    - There is no order if you didn’t define it.
- How do you define the natural order?

“Natural Ordering” & Comparable

- If a class implements the Comparable<*T* > interface
  - Objects from that class have a “natural ordering”
  - Objects can be compared **using** the compareTo() method
  - Collections.sort() can sort Lists of those objects automatically

- Comparable <*T* > is an Interface:
  - Requires
    - compareTo(*T* ob) → int
  - ob1.compareTo(ob2)
    - returns –ve if ob1 ordered before ob2
    - returns 0 if ob1 ordered with ob2
    - returns +ve if ob1 ordered after ob2

Making Face Comparable

```
public class Face implements Comparable<Face>{
        :
  public int size(){
      return (wd * ht);
    }
        :
/** Natural ordering is by size, small to large. */
    public int compareTo(Face other){
      if        ( this.size() < other.size() ) return  -1;
      else if ( this.size() > other.size() ) return  1;
      else  return 0;
    }
```

```
       :
    else if (button.equals("SmallToBig")){
   Collections.sort(crowd);
            for (Face f : crowd)
               f.render(canvas);
     }
```

Sorting with Comparators

- Suppose we need two different sorting orders at different times?
- Collections.sort(…) has two forms:
  - Sort by the natural order

    Collections.sort(todoList)
    - the values in todoList must be Comparable
  - Sort according to a specified order:

    Collections.sort(crowd, faceByArea)
    - faceByArea is a Comparator object for comparing the values in crowd.

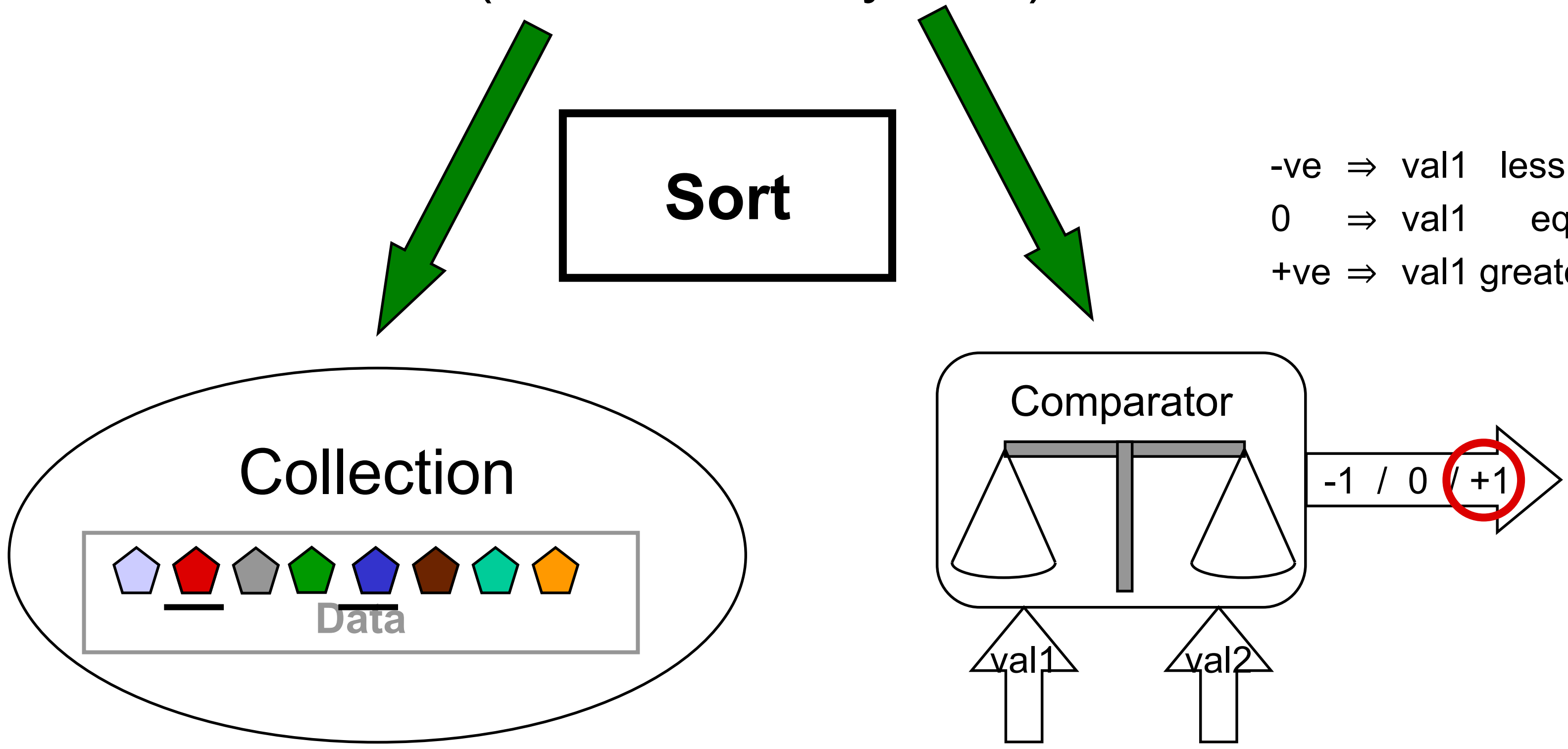
Comparable
interface for objects
that can be compared

⟺

Comparator
An object that can
compare other objects

Sorting with Comparators

- Collections.sort(crowd, faceByArea);



-ve	⇒	val1	less than	val2
0	⇒	val1	equals	val2
+ve	⇒	val1	greater than	val2

Comparators

- Comparator <T> is an Interface
- Requires
  - public int compare(T o1, T o2);

```
→ –ve  if o1 ordered before  o2
→   0  if o1       equals    o2     [ must be compatible with equals()! ]
→ +ve  if o1 ordered after   o2
```

```
/** Compares faces by the position of their top edge */
private class TopToBotComparator implements Comparator<Face>{
    public int compare(Face f1, Face f2){
        return (f1.getTop() - f2.getTop());
    }
}
```

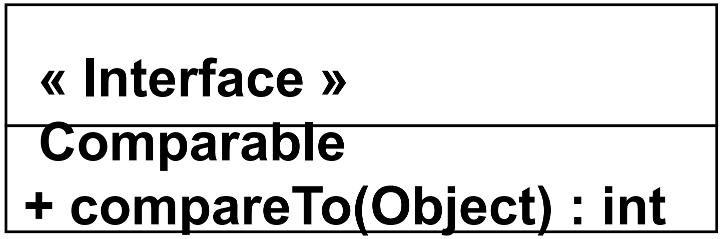
Using Multiple Comparators

```
String button = event.getActionCommand();
if (button.equals("SmallToBig")){
    Collections.sort(crowd);    // use the "natural ordering" on Faces.
    render();
}
else if (button.equals("BigToSmall")){
    Collections.sort(crowd, new BigToSmallComparator());
    render();
}
else if (button.equals("LeftToRight")){
    Collections.sort(crowd, new LftToRtComparator());
    render();
}
else if (button.equals("TopToBottom")){
    Collections.sort(crowd, new TopToBotComparator());
    render();
}
```

COMPARABLE VS COMPARATOR

- Classes should implement the **Comparable interface** to control their ***natural ordering*.**

- Objects that implement Comparable can be sorted by **Collections.sort()** and **Arrays.sort()** and can be used as keys in a sorted map or elements in a sorted set without the need to specify a *Comparator*.



- **compareTo()** compares this object with another object and returns a *negative* integer, *zero*, or a *positive* integer as this object is *less* than, *equal* to, or *greater* than the other object.

COMPARABLE VS COMPARATOR

- Use **Comparator** to sort objects in an order other than their natural ordering.

« Interface »
Comparator

+ compare(Object, Object) : int

- **compare()** compares its two arguments for order, and returns a *negative* integer, *zero*, or a *positive* integer as the first argument is *less* than, *equal* to, or *greater* than the second.

Q&A

- An object defined under a comparable class will have a “natural ordering”. (T or F)
- Objects declared under a comparable class can be compared using which method?
- What is the signature of the *compareTo* method?
- Which method can be used to sort list of comparable objects?
- Comparator is an object that can compare other objects. (T or F)
- What is the signature for the *compare()* method?
- A comparable class can implement multiple comparators. (T or F)

Summary

- Iterators and Iterables
- Sorting collections
- Comparators and Comparables