CMSC 433 – Programming Language Technologies and Paradigms Spring 2007

Inner Classes & Iterators

Mar. 6, 2007

Inner Classes

- Classes can be nested inside other classes
  - These are called *inner classes*
- Within a class that contains an inner class, you can use the inner class just like any other class

Example: The Queue Class

```
class Queue<Element> {
  class Entry { // Java inner class
    Element elt; Entry next;
    Entry(Element i) { elt = i; next = null; }
  }

  Entry theQueue;

  void enqueue(Element e) {
    if (theQueue == null) theQueue = new Entry(e);
    else {
      Entry last = theQueue;
      while (last.next != null) last = last.next;
      last.next = new Entry(e);
    }
  }
  ...
```


Example: The Queue Class (cont'd)

```
class Queue<Element> {
   ...
   Element dequeue() throws EmptyQueueException {
     if (theQueue == null)
       throw new EmptyQueueException();
     Element e = theQueue.elt;
     theQueue = theQueue.next;
     return e;
   }
}
```

Referring to Outer Class

```
class Queue<Element> {
 ...
 int numEntries;
 class Entry {
    Element elt; Entry next;
    Entry(Element i) { elt = i; next = null;
                       numEntries++; }
 }
}
```

- Each inner "object" has an implicit reference to the outer "object" whose method created it
  - Can refer to fields directly, or use outer class name.

Anonymous Inner Classes

```
(new Thread() {
       public void run() {
              try {
              Thread.sleep(1000*60*20);
              System.out.println("...");
              System.exit(1);
              } catch (Exception e) {}
       }
      }).start();
```

- Create anonymous subclass of thread, and invoke method on it

Other Features of Inner Classes

- Outside of the outer class, use outer.inner notation to refer to type of inner class
  - E.g., Queue.Entry
- An inner class marked *static* does not have a reference to outer class
  - Can't refer to instance variables of outer class
  - Must also use outer.inner notation to refer to inner class

Compiling Inner Classes

- The JVM doesn't know about inner classes
  - Compiled away, similar to generics
  - Inner class Foo of outer class A produces A$Foo.class
  - Anonymous inner class of outer class A produces A$1.class
- Why are inner classes useful?

Iteration

- Goal: Loop through all objects in an aggregate

```
class Node { Element elt; Node next; }
Node n = ...;
while (n != null) { ...; n = n.next; }
```

- Problems:
  - Depends on implementation details
  - Varies from one aggregate to another

Iterators in Java

```
public interface Iterator {
   // returns true if the iteration has more elts
  public boolean hasNext();

  // returns the next element in the iteration
  public Object next() throws NoSuchElementException;
}
```

(plus optional remove method)

- Implementation of aggregate not exposed
- Generic for wide variety of aggregates
- Supports multiple traversal strategies

Generic Iterators in Java 1.5

```
public interface Iterator<A> {
   // returns true if the iteration has more elts
  public boolean hasNext();

  // returns the next element in the iteration
  public A next() throws NoSuchElementException;
}
```


Using Iterators

```
import java.util.*;

public final class LoopStyles {

  public static void main( String[] aArguments ) {
    List<String> flavours = new ArrayList<String>();
    flavours.add("chocolate");
    flavours.add("strawberry");
    flavours.add("vanilla");

    useWhileLoop( flavours );

    useForLoop( flavours );
  }
```

Using Iterators (contd…)

```
private static void useWhileLoop( Collection<String> aFlavours )
   {
    Iterator<String> flavoursIter = aFlavours.iterator();
    while ( flavoursIter.hasNext() ){
      System.out.println( flavoursIter.next() );
    }
  }

  /**
  * Note that this for-loop does not use an integer index.
  */
  private static void useForLoop( Collection<String> aFlavours )
   {
    for ( Iterator<String> flavoursIter = aFlavours.iterator();
                                      flavoursIter.hasNext(); ) {
      System.out.println( flavoursIter.next() );
    }
  }
}
```


Iterators and Queues

- Recall queue example from beginning of lecture
- We'll explore options for adding iterators

next() Shouldn't Mutate Aggregate

```
class Queue<Element> {
  ...
  class QueueIterator implements Iterator<Element> {
    Entry rest;

    QueueIterator(Entry q) { rest = q; }
    boolean hasNext() { return rest != null; }
    Element next() throws NoSuchElementException {
      if (rest == null)
         throw new NoSuchElementException();
      Element e = rest.elt;
      rest = rest.next; // queue data intact
      return e;
      }
  }
}
```


Evil Mutating Clients

- But a client could mutate the data structure …

```
HashMap h = ...;
...
 //entrySet() Returns a collection view of the mappings
 //contained in this map.
Iterator i = h.entrySet().iterator();
System.out.println(i.next());
System.out.println(i.next());
 // put(Object key, Object value)
 //      Associates the specified value
 //with the specified key in this map.
h.put("Foo", "Bar"); // hash table resize!
System.out.println(i.next()); // prints ???
```

Defensive (Proactive) Copying

- Solution 1: Iterator copies data structure

```
class QueueIterator implements Iterator<Element> {
    Entry rest;

    QueueIterator(Queue q) {
      // copy q.theQueue to rest
    }
}
```

- Pro: Works even if queue is mutated
- Con: Expensive to construct iterator

Timestamps

- Solution 2: Track Mutations

```
class Queue<Element> {
  ...
  int modCount = 0;
  void enqueue(Element e) { ... modCount++; }
  Element dequeue() { ... modCount++; }

  ...
```


Timestamps (cont'd)

```
  ...
  class QueueIterator implements Iterator<Element> {
    int expectedModCount = modCount; // set at iterator
                                     // construction time

    Element next() {
      if (expectedModCount != modCount)
        throw new ConcurrentModificationException();
      ...
    }
    // does hasNext() need to be modified?
 }}
```

- Pro: Iteration construction cheap
- Con: Doesn't allow any mutation

What if Mutation is Allowed?

- Allowed mutation must be part of iterator spec

```
public void remove()
        throws IllegalStateException;
```

- 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.

Iterators

- Key ideas
  - Separate aggregate structure from traversal protocol
  - Support additional kinds of traversals
    - E.g., smallest to largest, largest to smallest, unordered
  - Multiple simultaneous traversals
    - Though many Java Collections do not provide this
- Structure
  - Iterator interface defines traversal protocol
  - Concrete Iterator implementations for each aggregate
    - And for each traversal strategy
  - Aggregate instances create Iterator object instances