Administrivia

- Reading, Liskov ch 6, 15
- Another resource: *Thinking in Patterns with Java*
  - Link from the class web page
- Project 2 due February 25
  - Version 3 of code posted

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

```java
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) {
        Entry last = theQueue;
        while (last.next != null) last = last.next;
        last.next = new Entry(e);
    }
    ...
}
```

Example: The Queue Class (cont’d)

```java
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

```java
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

```java
(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
• Question: Can Queue.Element be made static?

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
  ```java
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

```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

```java
public interface Iterator<? super 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

Iterator<Element> i = c.iterator();
while (i.hasNext()) {
    Element e = i.next();
    // do stuff with e
}

// alternatively use for
for (Iterator i = c.iterator(); i.hasNext(); ) {
    Element e = (Element) i.next();
    // do stuff with e
}

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 = ...;
    ...
    Iterator i = h.entrySet().iterator();
    System.out.println(i.next());
    System.out.println(i.next());
    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++; }
    ...
}
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

• Neither solution tracks mutations to container elts
  – Could use clone(), but tricky

• Allowed mutation must be part of iterator spec
  public void remove() { throw 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.

• 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

• Iterators are an example of a design pattern:
  – Design pattern = problem + solution in context
  – Iterators: solution for providing generic traversals

• Design patterns capture software architectures and designs
  – Not code reuse!
  – Instead, solution/strategy reuse
  – Sometimes, interface reuse

• The book that started it all
• Community refers to authors as the “Gang of Four”
• Figures and some text in these slides come from book
• On reserve in CS library (3rd floor AVW)
Object Modeling Technique (OMT)

- Used to describe patterns in GO4 book
- Graphical representation of OO relationships
  - **Class diagrams** show the static relationship between classes
  - **Object diagrams** represent the state of a program as series of related objects
  - **Interaction diagrams** illustrate execution of the program as an interaction among related objects

Classes

<table>
<thead>
<tr>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation1</td>
</tr>
<tr>
<td>Type: Operation2</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Object instantiation

```plaintext
Instantiator                      Instantiated
```

Subclassing and Abstract Classes

- **Concrete Subclass**
  - **Operation1**
  - ```plaintext
    ConcreteSubclass
    Operation1
    ```
  - **Implementation pseudo-code**

Pseudo-code and Containment

```plaintext
Window:
  Area(): 0
  width:
  height:

Rectangle:
  Area(): 0
  width:
  height:

return rectangle.Area() = Area()
return width * height
```

Object diagrams

```plaintext
aDrawing
  shape0
  shape1

aLineShape
aCircleShape
```
**Components of a Pattern**

- Name(s)
- Problem
  - Context
  - Real-world example
- Solution
  - Design/structure
  - Implementation
- Consequences
- Variations, known uses

**Iterator Pattern**

- Consequences:
  - Support different and simultaneous traversals
    - Multiple implementations of Iterator interface
    - One traversal per iterator instance
  - Requires coherent policy on aggregate updates
    - Invalidate iterator by throwing an exception, or
    - Iterator only considers elements present at the time of its creation
- Variations:
  - Internal vs. external iteration
    - Java Iterator is external

**Iterator Pattern, Again**

- Name: Iterator (aka Cursor)
- Problem:
  - How to process the elements of an aggregate in an implementation-independent manner?
- Solution:
  - Define an Iterator interface
    - `next()`, `hasNext()`, etc. methods
  - Aggregate returns an instance of an implementation of Iterator interface to control the iteration

**Internal Iterators**

```java
public interface InternalIterator<Element> {
    void iterate(Processor<Element> p);
}
public interface Processor<Element> {
    public void process(Element e);
}
```

- The internal iterator applies the processor instance to each element of the aggregate
  - Thus, entire traversal happens “at once”
  - Less control for client, but easier to formulate traversal

**Interaction diagrams**

**Structure of Iterator (Cursor) Pattern**