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

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

So---what’s a generic? Well, let me digress …
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();

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);
        }
    }
    ...
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;
    }
}

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());
    System.out.println(i.next()); // hash table resize!
    h.put("Foo", "Bar"); // prints ???

Defensive (Proactive) Copying

• Solution 1: Iterator copies data structure

    class QueueIterator extends Iterator<Element> {
        Entry rest;
        QueueIterator(Queue q) {
            // copy q.theQueue to rest
        }
    }

    Pro: Works even if queue is mutated
    Con: Expensive to construct iterator
• Solution 2: Track Mutations

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

• Pro: Iteration construction cheap
• Con: Doesn’t allow any mutation
Comments

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

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

Design Patterns

- 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
Gang of Four

- 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>ClassName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation1()</td>
</tr>
<tr>
<td>Type Operation2()</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instanceVariable1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type instanceVariable2</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Object instantiation

Instantiator  

Instantiatee
Subclassing and Abstract Classes

Pseudo-code and Containment
Object diagrams

Interaction diagrams
Structure of Iterator (Cursor) Pattern

```
Aggregate
  CreateIterator()

ConcreteAggregate
  CreateIterator()

return new ConcreteIterator(this)

Client

Iterator
  First()
  Next()
  Previous()
  CurrentItem()

ConcreteIterator
```