# The
Iterator
Pattern

The Iterator Pattern

- ## Intent

  - ⇨ Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation

  - ⇨ An *aggregate object* is an object that contains other objects for the purpose of grouping those objects as a unit. It is also called a *container* or a *collection*. Examples are a linked list and a hash table.

- ## Also Known As

  - ⇨ Cursor

- ## Motivation

  - ⇨ An aggregate object such as a list should allow a way to traverse its elements without exposing its internal structure

  - ⇨ It should allow different traversal methods

  - ⇨ It should allow multiple traversals to be in progress concurrently

  - ⇨ But, we really do not want to add all these methods to the interface for the aggregate
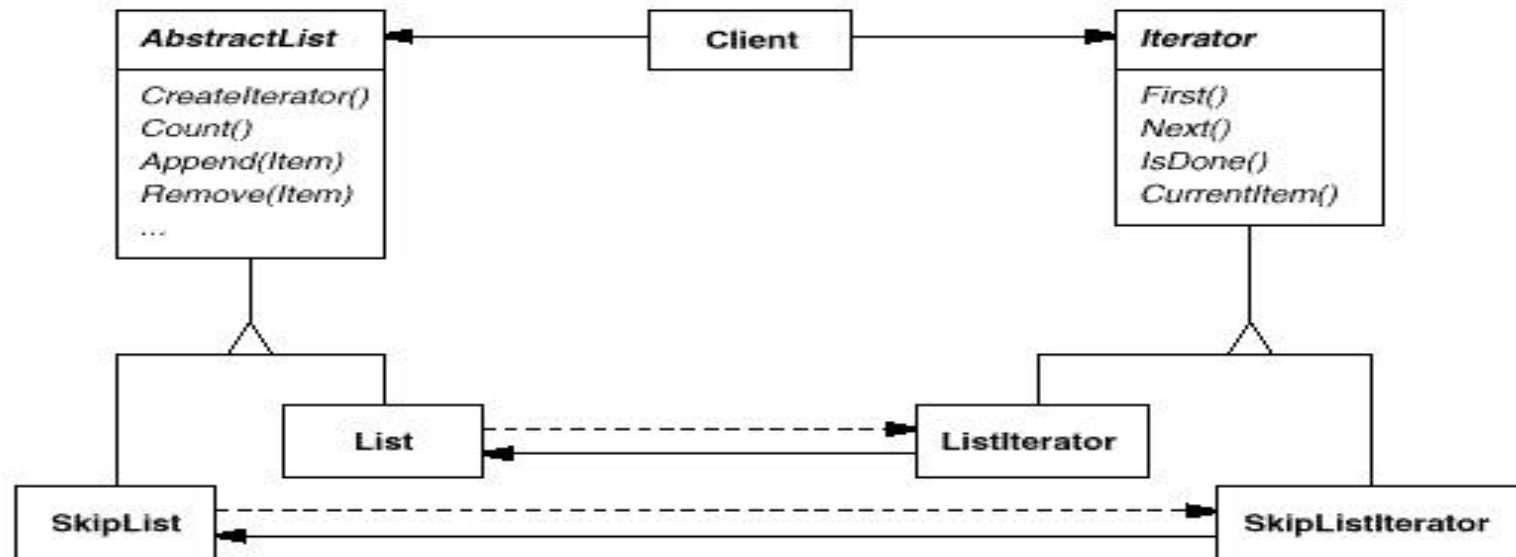
Iterator Example 1

- List aggregate with iterator:
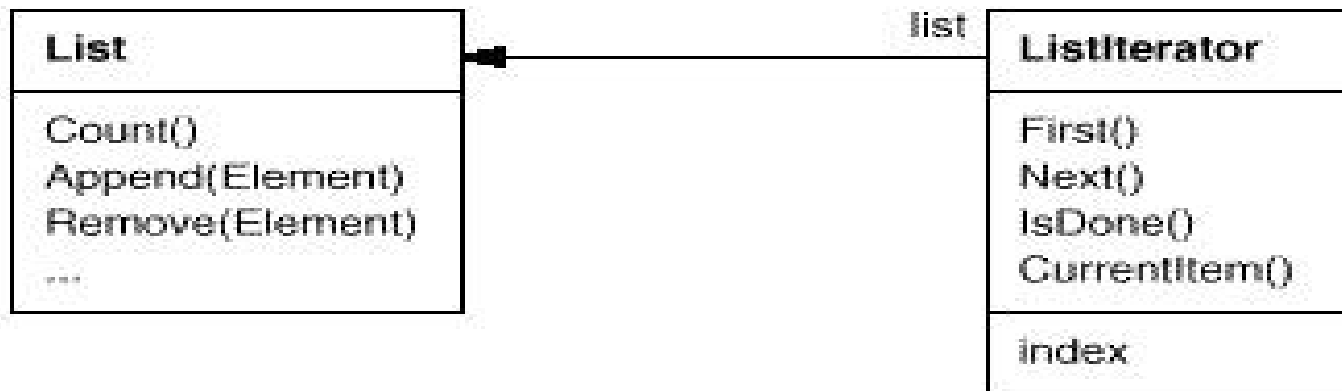
Iterator Example 1 (Continued)

- Typical client code:

```
...
List list = new List();
...
ListIterator iterator = new ListIterator(list);

iterator.First();
while (!iterator.IsDone()) {
  Object item = iterator.CurrentItem();
  // Code here to process item.
  iterator.Next();
}
...
```

Iterator Example 2

- Polymorphic Iterator

Iterator Example 2 (Continued)

- Typical client code:

```
List list = new List();
SkipList skipList = new SkipList();
Iterator listIterator = list.CreateIterator();
Iterator skipListIterator = skipList.CreateIterator();
handleList(listIterator);
handleList(skipListIterator);
...
public void handleList(Iterator iterator) {
  iterator.First();
  while (!iterator.IsDone()) {
    Object item = iterator.CurrentItem();
    // Code here to process item.
    iterator.Next();
  }
}
```
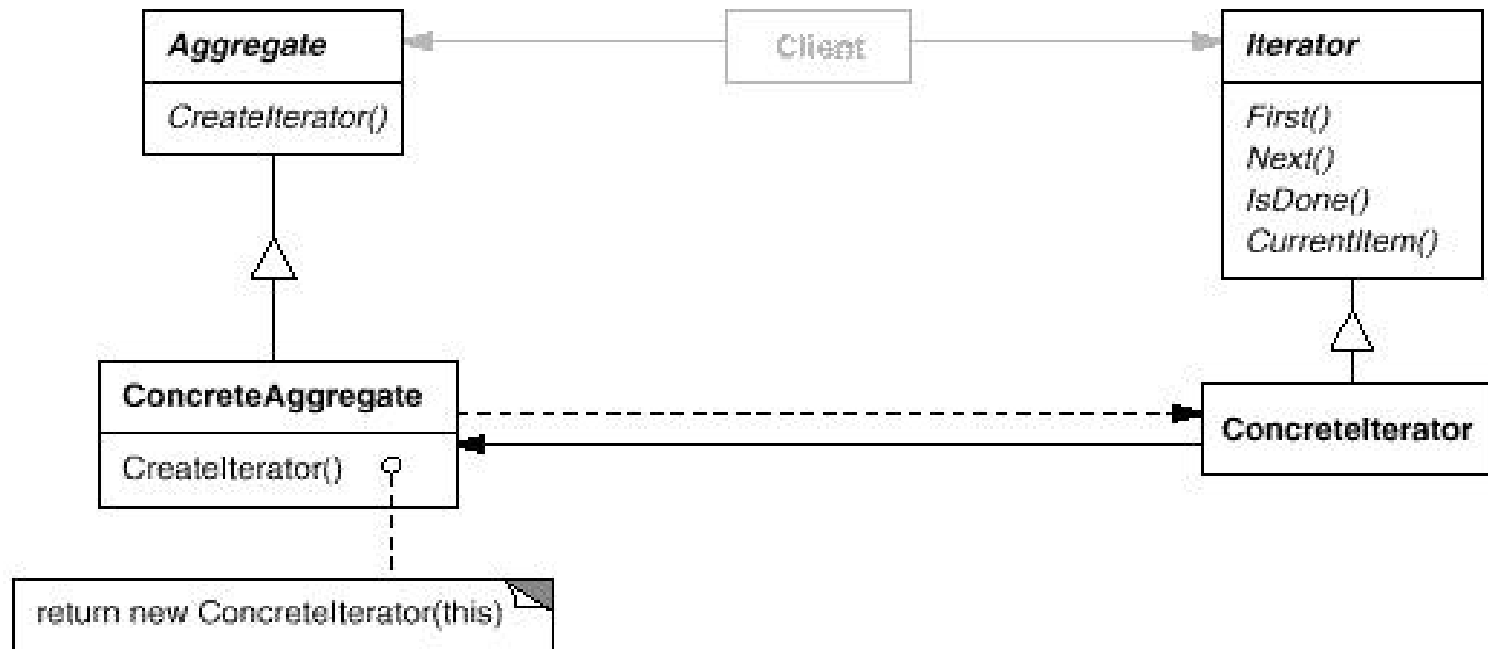
The Iterator Pattern

- ## Applicability

  Use the Iterator pattern:

  ⇨ To support traversals of aggregate objects without exposing their internal representation

  ⇨ To support multiple, concurrent traversals of aggregate objects

  ⇨ To provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration)

The Iterator Pattern

- Structure

The Iterator Pattern

- ● Participants
  - ⇨ Iterator
    - ➡ Defines an interface for accessing and traversing elements
  - ⇨ ConcreteIterator
    - ➡ Implements the Iterator interface
    - ➡ Keeps track of the current position in the traversal
  - ⇨ Aggregate
    - ➡ Defines an interface for creating an Iterator object (a factory method!)
  - ⇨ ConcreteAggregate
    - ➡ Implements the Iterator creation interface to return an instance of the proper ConcreteIterator

The Iterator Pattern

- ## Consequences

    - ⇨ Benefits

        - ➥ Simplifies the interface of the Aggregate by not polluting it with traversal methods

        - ➥ Supports multiple, concurrent traversals

        - ➥ Supports variant traversal techniques

    - ⇨ Liabilities

        - ➥ None!

The Iterator Pattern

- **Implementation Issues**

  ⇨ Who controls the iteration?

  - The client => more flexible; called an external iterator

  - The iterator itself => called an internal iterator

  ⇨ Who defines the traversal algorithm?

  - The iterator => more common; easier to have variant traversal techniques

  - The aggregate => iterator only keeps state of the iteration

  ⇨ Can the aggregate be modified while a traversal is ongoing? An iterator that allows insertion and deletions without affecting the traversal and without making a copy of the aggregate is called a *robust iterator*.

  ⇨ Should we enhance the Iterator interface with additional operations, such as previous()?

The Iterator Pattern

- ## Known Uses

  - ⇨ ObjectSpace's Java Generic Library containers

  - ⇨ Rogue Wave's Tools.h++ collections

  - ⇨ java.util.Enumeration interface

  - ⇨ Java 2 Collections Framework Iterator interface

- ## Related Patterns

  - ⇨ Factory Method

    - ➥ Polymorphic iterators use factory methods to instantiate the appropriate iterator subclass

  - ⇨ Composite

    - ➥ Iterators are often used to recursively traverse composite structures

Java Implementation Of Iterator

- We could implement the Iterator pattern "from scratch" in Java

- But, as was the case with the Observer pattern, Java provides built-in support for the Iterator pattern

- The java.util.Enumeration interface acts as the Iterator interface

- Aggregate classes that want to support iteration provide methods that return a reference to an Enumeration object

- This Enumeration object implements the Enumeration interface and allows a client to traverse the aggregate object

- Java JDK 1.1 has a limited number of aggregate classes: Vector and Hashtable

- Java JDK 1.2 introduced a new Collections package with more aggregate classes, including sets, lists, maps and an Iterator interface

The Enumeration Interface

- ## public abstract boolean hasMoreElements()

  - ⇨ Returns true if this enumeration contains more elements; false otherwise

- ## public abstract Object nextElement()

  - ⇨ Returns the next element of this enumeration

  - ⇨ Throws: NoSuchElementException if no more elements exist

- ## Correspondences:

  - ⇨ hasMoreElements() => IsDone()

  - ⇨ nextElement() => Next() followed by CurrentItem()

  - ⇨ Note that there is no First(). First() is done automatically when the Enumeration is created.

Enumeration Example

- Test program:

```
import java.util.*;

public class TestEnumeration {

  public static void main(String args[]) {
    // Create a Vector and add some items to it.
    Vector v = new Vector();
    v.addElement(new Integer(5));
    v.addElement(new Integer(9));
    v.addElement(new String("Hi, There!"));

    // Traverse the vector using an Enumeration.
    Enumeration ev = v.elements();
    System.out.println("\nVector values are:");
    traverse(ev);
```

Enumeration Example (Continued)

```java
// Now create a hash table and add some items to it.
Hashtable h = new Hashtable();
h.put("Bob", new Double(6.0));
h.put("Joe", new Double(18.5));
h.put("Fred", new Double(32.0));

// Traverse the hash table keys using an Enumeration.
Enumeration ekeys = h.keys();
System.out.println("\nHash keys are:");
traverse(ekeys);

// Traverse the hash table values using an Enumeration.
Enumeration evalues = h.elements();
System.out.println("\nHash values are:");
traverse(evalues);
}
```

```
private static void traverse(Enumeration e) {
  while (e.hasMoreElements()) {
    System.out.println(e.nextElement());
  }
}

}
```

Enumeration Example (Continued)

- Test program output:

```
Vector values are:

5

9

Hi, There!


Hash keys are:

Joe

Fred

Bob


Hash values are:

18.5

32.0

6.0
```

Vector Enumeration

- Let's take a look at Vector.java:

```java
public class Vector implements Cloneable, java.io.Serializable {
  ...
  protected Object elementData[];
  protected int elementCount;
  protected int capacityIncrement;

  public Vector(int initialCapacity, int capacityIncrement) {
    super();
    this.elementData = new Object[initialCapacity];
    this.capacityIncrement = capacityIncrement;
  }

  public Vector(int initialCapacity) {this(initialCapacity, 0);}
  public Vector() {this(10);}
```

- Vector.java (Continued):

```
public final synchronized Enumeration elements() {
  return new VectorEnumerator(this);
}

public final synchronized void addElement(Object obj) {
  int newcount = elementCount + 1;
  if (newcount > elementData.length) {
    ensureCapacityHelper(newcount);
  }
  elementData[elementCount++] = obj;
}
...
}
```

Vector Enumeration (Continued)

- Vector.java (Continued):

```java
final class VectorEnumerator implements Enumeration {
  Vector vector;
  int count;

  VectorEnumerator(Vector v) {
    vector = v;
    count = 0;
  }

  public boolean hasMoreElements() {
    return count < vector.elementCount;
  }
```

Vector Enumeration (Continued)

- Vector.java (Continued):

```java
public Object nextElement() {
    synchronized (vector) {
        if (count < vector.elementCount) {
            return vector.elementData[count++];
        }
    }
    throw new NoSuchElementException("VectorEnumerator");
    }
  }
...
}
```

- Do vector insertions affect the enumeration?? Yes! So the iterator provided by Vector is not robust.

The Java Collections Framework

- Java JDK 1.2 introduced a new Collections Framework to provide a unified architecture for representing and manipulating collections

- A collections framework contains three things:

  - Interfaces: abstract data types representing collections

  - Implementations: concrete implementations of the collection interfaces

  - Algorithms: methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces

The Java Collections Framework Interfaces

- Interfaces in the Java Collections Framework:
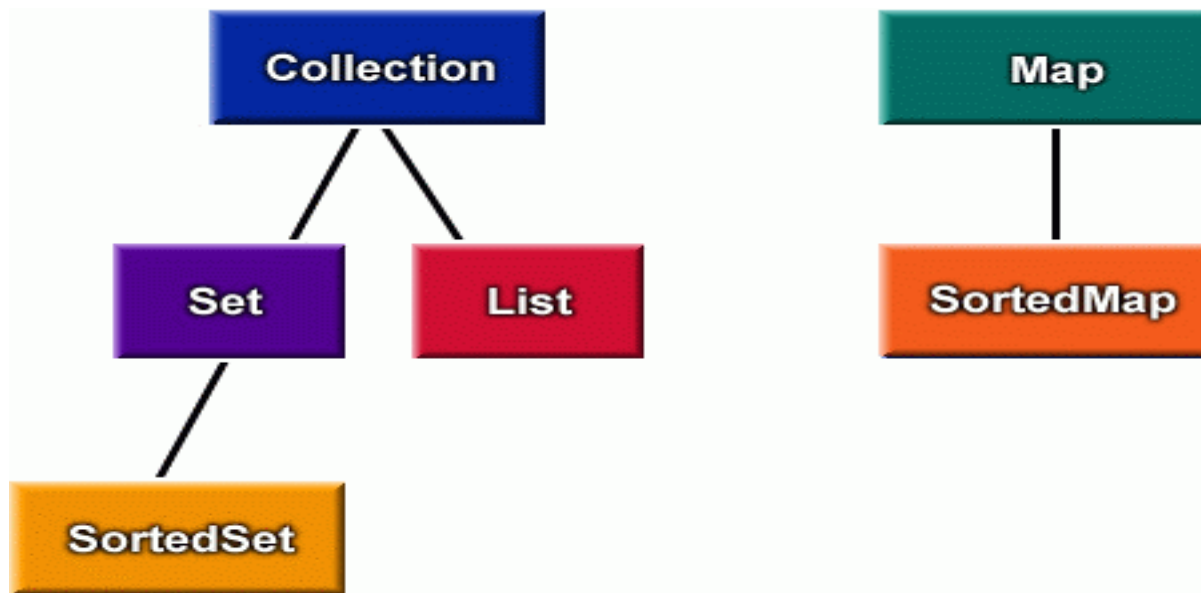
The Collection Interface

- The java.util.Collection interface:

```
public interface Collection {
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);
    boolean remove(Object element);
    Iterator iterator();
    ...
}
```

The Iterator Interface

- ## The Iterator object is similar to an Enumeration object with two differences:

  ⇨ Method names have changed:

  ➥ hasNext() instead of hasMoreElements()

  ➥ next() instead of nextElement()

  ⇨ Iterator is more robust than Enumeration in that it allows (with certain constraints) the removal (and in some cases addition) of elements from the underlying collection during the iteration

- ## The java.util.Iterator interface:

```
public interface Iterator {
   boolean hasNext();
   Object next();
   void remove();
}
```

The List Interface

- java.util.List interface:

```java
public interface List extends Collection {
    Object get(int index);
    Object set(int index, Object element);
    void add(int index, Object element);
    Object remove(int index);
    boolean addAll(int index, Collection c);
    int indexOf(Object o);
    int lastIndexOf(Object);
    ListIterator listIterator();
    ListIterator listIterator(int index);
    List subList(int from, int to);
}
```

The ListIterator Interface

- java.util.ListIterator interface:

```
public interface ListIterator extends Iterator {
   boolean hasNext();
   Object next();
   boolean hasPrevious();
   Object previous();
   int nextIndex();
   int previousIndex();
   void remove();
   void set(Object o);
   void add(Object o);
}
```

LinkedList Example

- The java.util.LinkedList class is a concrete class that implements the List interface

- Here's an example using LinkedList:

```
List list = new LinkedList();
...
ListIterator iterator = list.listIterator();
...
while (iterator.hasNext()) {
  Object item = iterator.next();
  // Code here to process item.
}
```

LinkedList Example (Continued)

- You can even traverse the list backwards:

```
List list = new LinkedList();

...

ListIterator iterator = list.listIterator(list.size());

...

while (iterator.hasPrevious()) {
  Object item = iterator.previous();
  // Code here to process item.
}
```

LinkedList Example (Continued)

- And you can remove items as you traverse!

```
List list = new LinkedList();
...
ListIterator iterator = list.listIterator();
...
while (iterator.hasNext()) {
  Object item = iterator.next();
  // If element satisfies some condition,
  //   remove it using the iterator, else
  //   process it.
  if (testOnItem(item))
    iterator.remove();
  else
    // Code here to process item.
}
```

LinkedList Example (Continued)

- The remove method of Iterator removes the last element that was returned by next(). The remove() method may be called only once per call to next(), and throws an exception if this condition is violated.

- Using Iterator.remove() is the only safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress and, in fact, an exception will be thrown

The LinkedList Class

- Let's take a look at LinkedList.java to see how the robustness is implemented:

```
/**
 * Linked list implementation of the List interface.
 * Note that this implementation is not synchronized. If multiple
 * threads access a list concurrently, and at least one of the
 * threads modifies the list structurally, it must be synchronized
 * externally. This is typically accomplished by synchronizing on
 * some object that naturally encapsulates the list or by wrapping
 * the list using the Collections.synchronizedList method:
 *    List list = Collections.synchronizedList(new LinkedList(...));
 */
public class LinkedList extends AbstractSequentialList
            implements List, Cloneable, java.io.Serializable {
  private transient Entry header = new Entry(null, null, null);
  private transient int size = 0;
  ...
```

The LinkedList Class (Continued)

```java
/**
 * Returns a list-iterator of the elements in this list (in proper
 * sequence), starting at the specified position in the list.
 * The list-iterator is fail-fast: if the list is structurally
 * modified at any time after the Iterator is created, in any way
 * except through the list-iterator's own remove or add methods,
 * the list-iterator will throw a ConcurrentModificationException.
 * Thus, in the face of concurrent modification, the iterator
 * fails quickly and cleanly, rather than risking arbitrary,
 * non-deterministic behavior at an undetermined time in the
 * future.
 */
public ListIterator listIterator(int index) {
  return new ListItr(index);
}
```

The LinkedList Class (Continued)

```
/**
 * Private class that provides the implementation for the
 * iterator.
 */
private class ListItr implements ListIterator {
  private Entry lastReturned = header;
  private Entry next;
  private int nextIndex;
  private int expectedModCount = modCount;

  // Note: modCount is inherited from AbstractSequentialList.
  // It is incremented each time the list is modified.
  ...
  public boolean hasNext() {
    return nextIndex != size;
  }
}
```

The LinkedList Class (Continued)

```java
public Object next() {
  checkForComodification();
  if (nextIndex == size)
    throw new NoSuchElementException();
  lastReturned = next;
  next = next.next;
  nextIndex++;
  return lastReturned.element;
}

final void checkForComodification() {
  if (modCount != expectedModCount)
    throw new ConcurrentModificationException();
}
```

The Iterator Pattern
36

The LinkedList Class (Continued)

```java
    public void remove() {
      LinkedList.this.remove(lastReturned);
      if (next==lastReturned)
        next = lastReturned.next;
      else
        nextIndex--;
      lastReturned = header;
      expectedModCount++;
    }
  ...
  }
  ...
}
```