Iterator Pattern: Motivation

- Suppose you have two classes, each of which implements a collection of items of some sort
  - Sample code:
    ```java
    public class RepOne {
        ArrayList itemList;

        public RepOne (...) {
            itemList = new ArrayList();
            ...
        }

        public void addItem (args) {
            ListItem listItem = new ListItem(args);
            itemList.add(listItem);
        }
        ...
    }
    
    public class RepTwo {
        int listSize;
        int numElements = 0;
        ListItem[] itemList;

        public RepTwo (...) {
            itemList = new ListItem[listSize];
            ...
        }

        public void addItem (args) {
            ListItem listItem = new ListItem(args);
            if (noOfItems < listSize) {
                listItem[numElements] = new ListItem(args);
                numElements++;  
            }
        }
        ...
    }
    
    - While the representation for the collections differs between the implementations, the items in the collections are of the same type
Iterator Pattern: Motivation

- If we have a client that needs to iterate over each kind of collection, we would need separate pieces of code to do this:

```
//Client code

ListOne listOne = new ListOne(args);
ListTwo listOne = new ListTwo(args);

//Add items to lists
for (int i = 0; i < listOne.size(); i++) {
    ListItem listItem = (ListItem)listOne.get(i);
    //process item
}
for (int i = 0; i < listTwo.length; i++) {
    ListItem listItem = listTwo[i];
    //process item
}
```

- The problem with this approach is that the client
  1. Needs a separate loop to process each collection;
  2. Needs knowledge of the implementation of each collection;
  3. If a new collection implementation is accessed by the client, you would need to modify existing client code.

- Our rule of thumb is to *encapsulate what varies*
  - In this case, it’s the iteration that varies
  - Goal is to be able to
    1. Iterate over the items in each collection using a *single* implementation
      * Do *NOT* want to recode one or the other of the two classes
      * I.e., you would like the implementation of the collections to be transparent to the iteration
    2. This common interface should support
      (a) Accessing elements of a collection regardless of the collection implementation details
      (b) Determining whether there are elements yet to be accessed
      (c) Accessing the next element in the collection if there are still items not visited
Iterator Pattern: Defined

- **Iteration Pattern:**
  
  *Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.*

- **Class diagram:**

- **Methods:**
  
  - `hasNext()` indicates whether there remain any collection items to be processed
  
  - `next()` returns the next element in the collection
  
  - `remove()` deletes the last item returned by `next()` from the collection
    
    * Considered optional for the pattern
  
- **Advantages:**
  
  - Provides a uniform means of accessing elements regardless of the implementation
  
  - Moves responsibility out of the client and into the iterator

- **Refered to as *polymorphic* iteration**
  
  - Means that use of polymorphism allows client to ignore implementation of collection being iterated over
Iterator Pattern: Revised Example

• Sample code:

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

public interface ListInterface {
    public Iterator createIterator();
}

public class RepOneIterator implements Iterator{
    ArrayList itemList;
    int position = 0;

    public RepOneIterator (ArrayList itemList) {
        this.itemList = itemList;
    }

    public Object next() {
        ListItem nextItem = itemList.get(position);
        position++;
        return nextItem;
    }

    public boolean hasNext () {
        if (position < itemList.Size())
            return false;
        else
            return true;
    }

    public remove () {
        itemList.remove(position); *****
    }
}
```
public class RepTwoIterator implements Iterator{
    ListItem[] itemList;
    int position = 0;

    public RepTwoIterator (ListItem[] itemList) {
        this.itemList = itemList;
    }

    public Object next() {
        ListItem nextItem = itemList[position];
        position++;
        return nextItem;
    }

    public boolean hasNext () {
        if (position < itemList.getNumElements())
            return false;
        else
            return true;
    }

    public remove () {
        itemList.deleteElement(position);
    }
}

public class RepOne implements ListInterface{
    ArrayList itemList;

    public RepOne (...) {
        itemList = new ArrayList();
        ...
    }

    public void addItem (args) {
        ListItem listItem = new ListItem(args);
        itemList.add(listItem);
    }

    public Iterator createIterator () {
        return new RepOneIterator(itemList);
    }

    ...
}
public class RepTwo implements ListInterface{
    int listSize;
    int numElements = 0;
    ListItem[] itemList;

    public RepTwo (...) {
        itemList = new ListItem[listSize];

        ...
    }

    public void addItem (args) {
        ListItem listItem = new ListItem(args);
        if (noOfItems < listSize) {
            listItems[numElements] = new ListItem(args);
            numElements++;
        }
    }

    public Iterator createIterator () {
        return new RepTwoIterator(itemList);
    }

    ...
}

public class Driver {
    public static void main (String args[]) {
        ListOne firstList = new ListOne();
        ListTwo secondList = new ListTwo(10);
        ListOneIterator firstIterator = new ListOneIterator(firstList);
        ListTwoIterator secondIterator = new ListTowIterator(secondList);

        //add some items to the lists
        processList(firstList);
        processList(secondList);
    }

    public processList (Iterator iterator) {
        while (iterator.hasNext) {
            ListItem listItem = (ListItem)iterator.next();
           ..
        }
    }

    ...
}
Iterator Pattern: Java’s Implementation

- Java’s interface for iterators found in java.util.Iterator
- Is exactly the same as interface developed above
- Java Collections include a method for creating an Iterator for the class
- Changes needed in above code:
  1. Add `import java.util.Iterator;` to `Driver`
  2. Delete `Iterator` interface
  3. Change body of `createIterator()` in `RepOne` to `return itemList.iterator();`
     - Are now using built-in functionality of `Collection`
- Variations
  - Original `Iterator` methods were
    1. `next()`: Same as above
    2. `isDone()`: corresponds to `hasNext()`
    3. `first()`: Nothing equivalent
       * Not considered that useful as usually create a new iterator when want to reiterate
  - Internal v external iterator
     * External
       · Client controls iteration using `next()` (as above)
     * Internal
       · Traversal of list controlled from within iterator itself
       · Requires an operation as a parameter that processes the elements as they are accessed
       · Not as flexible as external
  - Can implement so can move in either direction
  - Iteration implies no intrinsic order
  - `Enumeration` interface is a precursor to `Iterator`
     * Original `Enumeration` methods were
       1. `nextElement()`: Same as `next()`
       2. `hasMoreElements()`: corresponds to `hasNext()`
     * `Iterator”better”` as supported by more classes
Iterator Pattern: Design Principle of Single Responsibility

- **Design Principle 9 (Single Responsibility)**
  
  *A class should have only one reason to change.*

- If a class assumed more than one responsibility (e.g., managing a data structure and iterating over that structure), then there is a greater possibility for modification
  
  - The greater the number of changes, the greater the chance for error

- **Cohesion**
  
  - Measures how closely a module supports a single purpose
  - *High* cohesion is when it is designed around a set of related functions
  - *Low* cohesion is when it is designed around a set of unrelated functions

- Cohesion similar to *Single Responsibility*, but more general
  
  - Generally, classes that adhere to *Single Responsibility* tend to exhibit high cohesion
Iterator Pattern: Java Collections Framework

• Provides a set of frameworks and classes for representing aggregates

• Interfaces (and corresponding classes) include
  – Set Interface
    * HashSet
    * EnumSet
    * TreeSet
  – List Interface
    * ArrayList
    * LinkedList
    * Vector
    * Stack
  – Map Interface
    * HashMap
    * TreeMap
    * Hashtable
  – Queue Interface
  – BlockingQueue Interface

• Collection interface provides the following:
  – add()
  – addAll()
  – clear()
  – contains()
  – containsAll()
  – equals()
  – hashCOde()
  – isEmpty()
  – iterator()
  – remove()
  – removeAll()
  – retainAll()
  – size()
  – toArray()
Iterator Pattern: Iteration in Java 5

• Java 5 provides a variant of the *for* loop for iterating over collections and arrays

• Syntax:

  ```java
  for (Object object: aggregate) {
    //process object
  }
  ```

• The *for* loop will iterate over the elements in *aggregate*, returning each one in *object*

• For the *RepOne* class from the earlier example:

  ```java
  for (listItem item: firstList) {
    //process item
  }
  ```