Java Collections Framework: Interfaces

• Introduction to the Java Collections Framework (JCF)
• The Comparator Interface Revisited
• The Collection Interface
• The List Interface
• The Iterator Interface
• The ListIterator Interface
The Java Collections Framework

- A **collection** is a group of objects.
- The **Java Collections Framework** (JCF) is a set of important utility classes and interfaces in the java.util package for working with collections.
- The Java Collections Framework consists of three parts:
  - Interfaces: the abstract data types that the framework supports.
    - E.g., java.util.Comparator, java.util.Collection, java.util.Iterator
  - Implementations: concrete versions of these interfaces.
    - E.g., java.util.ArrayList and java.util.LinkedList.
  - Algorithms: predefined actions defined on the interfaces or their implementations. E.g., Collections.sort(), Collections.binarySearch()
Why Develop the JCF?

• An array is a common data structure suitable in many situations
• However, arrays are not always good for some of the following:
  – They require size information for creation
  – Inserting an element in the middle of an array leads to moving other elements around
  – Deleting an element from the middle of an array leads to moving other elements around
• Other data structures, like linked list and trees, are better for some situations
• The JCF provides efficient implementations for these common data structures
• By using the JCF interfaces, these data structures can be manipulated in a uniform way
Some Benefits of using the JCF

• Benefits
  – Reduces programming effort: by providing useful data structures, the programmer can focus on the problem at hand.
  – Improves program speed and quality: by providing high-performance, high-quality implementations for the most common data structures
  – Allows interoperability among unrelated APIs
  – Fosters software reuse

• Concerns
  – JCF data structures work only with objects, not primitive types. If we deal with primitive types, we use Wrapper classes.
  – A Collection can contain incompatible types at the same time
  – JFC methods are generic; must always downcast from Object to our types
The Collection interface

- Root of the collection hierarchy
- Represents a group of objects, known as its *elements*.
- Some *Collection* implementations allow duplicate elements and others do not. Some are ordered and others unordered.
- Defines all of the operations that are common to most if not all kinds of collections
- Some collection *operations* are *optional* – meaning that some implementations may not support them. In practice such methods would throw a *runtime UnsupportedOperation* exception
The Collection Interface

// Basic Operations
int size();
boolean isEmpty();
boolean contains(Object element);
boolean add(Object element);
boolean remove(Object element);
Iterator iterator();

// Bulk Operations
boolean containsAll(Collection c);
boolean addAll(Collection c);
boolean removeAll(Collection c);
boolean retainAll(Collection c);
void clear(); // Optional

// Array Operations
Object[] toArray();
Object[] toArray(Object a[]);
The List Interface

- The List interface represents an ordered collection of objects.
- Each element in a list has an index, or position. The indexes range from 0 to `size() – 1`.
- List extends Collection. It has the following additional methods to those it inherits and overrides:

```java
public abstract void add(int index, Object object);
public abstract Object get(int index);
public abstract int indexOf(Object object);
public abstract int lastIndexOf(Object object);
public abstract Object remove(int index);
public abstract Object set(int index, Object object);
public abstract ListIterator listIterator();
public abstract ListIterator listIterator(int index);
public abstract List subList(int fromIndex, int toIndex);
```
The Iterator Interface

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

- Iterators provide a way of visiting the elements of a collection one by one.
- For arrays, the loop index represents the iterator.
  ```java
  for (int i=0; i < 10; i++) {
  }
  ```
- Think of an iterator as a cursor that resides between elements of the collection.
- To visit all the elements of a collection object `c`, a code such as the following may be used:
  ```java
  Iterator iter = c.iterator();
  while(iter.hasNext()){  
    Object obj = iter.next();
    process(obj);
  }
  ```
## The Iterator Interface (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object next()</td>
<td>Moves the iterator forward one position and returns the element it has passed over</td>
</tr>
<tr>
<td>void remove()</td>
<td>The remove() method of an iterator removes the element whose reference was returned by the last call to next().</td>
</tr>
<tr>
<td>boolean hasNext()</td>
<td>Returns true if the iterator can be moved forward</td>
</tr>
</tbody>
</table>

![Diagram](image)
The Iterator Interface (cont’d)

• The following code removes the first element in a collection c:

```
Iterator iter = c.iterator();
iter.next();       // skip over the first element
iter.remove();    // remove the first element
```

• It is illegal to call the remove() method of an iterator if it was not preceded by a call to next(). For example, the following is invalid. Each remove call has to be preceded by a call to next().

```
Iterator iter = c.iterator(); or
iter.remove();    // illegal
```

Executing such program fragments will lead to run time exception IllegalstateException
Example 1: The Iterator Interface

```java
1 import java.util.*;
2 public class TestIterator {
3    public static void main(String[] args) {
4        // LinkedList list = new LinkedList(); // making arraylist or linkedlist is different at this
5        ArrayList list = new ArrayList(); // level. The remaining statements are the same
6        for (int i = 0; i < 6; i++)
7           list.add(new Integer(i)); // add method of list object
8        Iterator iter = list.iterator();
9        // find the sum of the integer values stored in list
10       int sum = 0;
11       while (iter.hasNext()) {
12          Object item = iter.next(); // Next item as an Object.
13          Integer num = (Integer)item; // Type-cast item to an Integer.
14          sum = sum + num.intValue(); // Add the number to the sum
15       }
16       System.out.println(list);
17       System.out.println("sum = \"+sum);
18    }
19 }
```
The ListIterator Interface

- The ListIterator interface extends Iterator to allow bi-directional traversal of a list, and the modification of a list.
- It has the following additional methods to those it inherits and overrides:

  ```java
  public abstract boolean hasPrevious();
  public abstract int nextIndex();
  public abstract Object previous();
  public abstract int previousIndex();
  public abstract add(Object object);
  public abstract void set(Object object);
  ```

- `listIterator()` or `listIterator(int pos)` are the methods that return an instance of `ListIterator`. `listIterator()` returns an iterator at position 0, but `listIterator(int pos)` returns an iterator that starts at position pos.
- ListIterator supports reverse traversal via previous and hasPrevious
- In ListIterator, the cursor sits between the element that would be returned by a call to next() and the element that would be returned by a call to previous().
The ListIterator interface (continued)

A description and some conditions for set, add, and remove methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>Overwrites last element returned by next() or previous(). NB: set must be preceded by a call to next() or previous()</td>
</tr>
<tr>
<td>add</td>
<td>Adds immediately before the current cursor position (ie in between elements that would be returned by previous() and next())</td>
</tr>
<tr>
<td>remove</td>
<td>Removes last element returned by next() or previous(). NB: remove must be preceded by a call to next() or previous()</td>
</tr>
</tbody>
</table>
Example 2: The ListIterator Interface

```java
import java.util.*;
public class TestListIterator {
    public static void main(String[] args) {
        ArrayList list = new ArrayList();

        ListIterator iter2, iter1 = list.listIterator();

        for(int i = 0; i < 6; i++)
            iter1.add(new Integer(i));  // add method of list iterator

        iter2 = list.listIterator(list.size()); // places iterator at the end
        iter2.previous();     // jumps over the last element in reverse order
        iter2.add(new Integer(25));  // add immediately before the last element
        iter2.previous();
        iter2.remove();
        System.out.println(list);
    }
}
```
Checking Iterators

• The ability of iterators to modify a collection can cause problems if there are several iterators attached to one collection.

• Assume that list has many objects. The last statement of this code fragment generates an exception

```
ListIterator iter1 = list.listIterator();
ListIterator iter2 = list.listIterator(); // two iterators for the same list
iter1.next();
iter1.remove();
iter2.next(); // throws ConcurrentModificationException
```

• Follow this rule:
  – You can have many read-only iterators, attached to a collection.
  – You can have just one read/write iterator attached to a collection.
Example 3: The ListIterator Interface

1 import java.util.*;
2 public class TestListIterator2 {
3    public static void main(String[] args) {
4       ArrayList list = new ArrayList();
5       int bonus = 1;
6
7       for(int i = 0; i < 6; i++)
8          list.add(new Integer(i));  // add method of the list
9
10      ListIterator iter = list.listIterator();
11
12      System.out.println("List before: " + list);
13
14      while (iter.hasNext()){
15          Integer myInt = (Integer)iter.next();
16          iter.set(new Integer(myInt.intValue()+bonus)); // replacing the object returned by
17              // next() by the one between brackets of set method)
18      }
19      System.out.println("List after : " + list);
20   }
21 }