JAVA COLLECTION FRAMEWORK & SETS

Ch07.4-5 & Ch10.5

Presentation for use with the textbook
We learned about part of the **List** ADT

The classes that implement the **List** interface are all *indexed* collections

- An index or subscript is associated with each element
- The element's index often reflects the relative order of its insertion into the list
- Searching for a particular value in a list is generally $O(n)$
- An exception is a binary search of a sorted object, which is $O(\log n)$
LIST ADT REVIEWED

/** A simplified version of the java.util.List interface. */

public interface List<E> {
    /** Returns the number of elements in this list. */
    int size();

    /** Returns whether the list is empty. */
    boolean isEmpty();

    /** Returns (but does not remove) the element at index i. */
    E get(int i) throws IndexOutOfBoundsException;

    /** Replaces the element at index i with e, and returns the replaced element. */
    E set(int i, E e) throws IndexOutOfBoundsException;

    /** Inserts element e to be at index i, shifting all subsequent elements later. */
    void add(int i, E e) throws IndexOutOfBoundsException;

    /** Removes/returns the element at index i, shifting subsequent elements earlier. */
    E remove(int i) throws IndexOutOfBoundsException;
}

Code Fragment 7.1: A simple version of the List interface.
JAVA.UTIL.LIST INTERFACE AND ITS IMPLEMENTERS
## Methods of the `ArrayList` Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public E get(int index)</code></td>
<td>Returns a reference to the element at position <code>index</code>.</td>
</tr>
<tr>
<td><code>public E set(int index, E anEntry)</code></td>
<td>Sets the element at position <code>index</code> to reference <code>anEntry</code>. Returns the previous value.</td>
</tr>
<tr>
<td><code>public int size()</code></td>
<td>Gets the current size of the <code>ArrayList</code>.</td>
</tr>
<tr>
<td><code>public boolean add(E anEntry)</code></td>
<td>Adds a reference to <code>anEntry</code> at the end of the <code>ArrayList</code>. Always returns true.</td>
</tr>
<tr>
<td><code>public void add(int index, E anEntry)</code></td>
<td>Adds a reference to <code>anEntry</code>, inserting it before the item at position <code>index</code>.</td>
</tr>
<tr>
<td><code>int indexOf(E target)</code></td>
<td>Searches for <code>target</code> and returns the position of the first occurrence, or -1 if it is not in the <code>ArrayList</code>.</td>
</tr>
<tr>
<td><code>public E remove(int index)</code></td>
<td>Returns and removes the item at position <code>index</code> and shifts the items that follow it to fill the vacated space.</td>
</tr>
</tbody>
</table>

`public boolean isEmpty()`
### The LinkedList Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void add(int index, E obj)</td>
<td>Inserts object obj into the list at position index.</td>
</tr>
<tr>
<td>public void addFirst(E obj)</td>
<td>Inserts object obj as the first element of the list.</td>
</tr>
<tr>
<td>public void addLast(E obj)</td>
<td>Adds object obj to the end of the list.</td>
</tr>
<tr>
<td>public E get(int index)</td>
<td>Returns the item at position index.</td>
</tr>
<tr>
<td>public E getFirst()</td>
<td>Gets the first element in the list. Throws NoSuchElementException if the list is empty.</td>
</tr>
<tr>
<td>public E getLast()</td>
<td>Gets the last element in the list. Throws NoSuchElementException if the list is empty.</td>
</tr>
<tr>
<td>public boolean remove(E obj)</td>
<td>Removes the first occurrence of object obj from the list. Returns true if the list contained object obj; otherwise, returns false.</td>
</tr>
<tr>
<td>public int size()</td>
<td>Returns the number of objects contained in the list.</td>
</tr>
</tbody>
</table>

**public boolean isEmpty()**

Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010
An iterator is a software design pattern that abstracts the process of scanning through a sequence of elements, one element at a time.

An Iterator object for a list starts at the first node.

The programmer can move the Iterator by calling its next method.

The Iterator stays on its current list item until it is needed.

An Iterator traverses in $O(n)$ while a list traversal using get() calls in a linked list is $O(n^2)$.
The **List** interface declares the method **iterator** which returns an **Iterator** object that iterates over the elements of that list.

Java defines the **java.util.Iterator** interface with the following two methods:

- **hasNext()**: Returns true if there is at least one additional element in the sequence, and false otherwise.
- **next()**: Returns the next element in the sequence.
- **remove()**: Removes from the collection the element returned by the most recent call to next(). Throws an **IllegalStateException** if next has not yet been called, or if remove was already called since the most recent call to next.
An **Iterator** is conceptually _between_ elements; it does not refer to a particular object at any given time.

A single iterator instance **supports only one pass through a collection**

+ there is no way to “reset” the iterator back to the beginning of the sequence.
EXAMPLE OF IMPLEMENTING ITERATOR

- Nested ArrayIterator Class inside ArrayList

```java
/**
 * A (nonstatic) inner class. Note well that each instance contains an implicit
 * reference to the containing list, allowing it to access the list's members.
 */

private class ArrayIterator implements Iterator<E> {
    private int j = 0;  // index of the next element to report

    public boolean hasNext() {
        return j < size;
    }

    public E next() throws NoSuchElementException {
        return (E) array[j++];
    }

    public void remove() throws IllegalStateException {
        if (j == 0)
            throw new IllegalStateException;
        array[j - 1] = null;
        j--;
    }

    /** Returns an iterator of the elements stored in the list. */
    public Iterator<E> iterator() {
        return this;
    }
}
```

- Additional Method of ArrayList

© 2014 Goodrich, Tamassia, Goldwasser
To provide greater standardization, Java defines another parameterized interface, named `Iterable`, that includes the following single method:

```
iterator(): Returns an iterator of the elements in the collection.
```

Each call to `iterator()` returns a new iterator instance, thereby allowing multiple (even simultaneous) traversals of a collection.

An instance of a typical collection class in Java, such as an `ArrayList`, is `iterable` (but not itself an `iterator`);
**While Loop with Iterator**

```java
Iterator<ElementType> iter = collection.iterator();
while (iter.hasNext()) {
    ElementType variable = iter.next();
    // Do something with value
}
```

**EX> process all items in List<Integer> through an Iterator**

```java
Iterator<Integer> iter = aList.iterator();
while (iter.hasNext()) {
    int value = iter.next();
    // Do something with value
}
```
You can use the `Iterator remove()` method to remove items from a list as you access them.

- `remove()` deletes the most recent element returned.

You must call `next()` before each `remove()`; otherwise, an `IllegalStateException` will be thrown.

- `LinkedList.remove` vs. `Iterator.remove`:
  - `LinkedList.remove` must walk down the list each time, then remove, so in general it is $O(n^2)$.
  - `Iterator.remove` removes items without starting over at the beginning, so in general it is $O(n)$. 

Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010
To remove all elements from a list of type `Integer` that are divisible by a particular value:

```java
public static void removeDivisibleBy(LinkedList<Integer> aList, int div) {
    Iterator<Integer> iter = aList.iterator();
    while (iter.hasNext()) {
        int nextInt = iter.next();
        if (nextInt % div == 0) {
            iter.remove();
        }
    }
}
```
Java 5.0 introduced an enhanced for statement

The enhanced for statement creates an Iterator object and implicitly calls its hasNext and next methods

Other Iterator methods, such as remove, are not available
The while statement in the “for-each” loop syntax

```java
for (ElementType variable : collection) {
    loopBody
} // may refer to "variable"
```

The following code counts the number of times `target` occurs in `myList` (type `LinkedList<String>`)
In list `myList` of type `LinkedList<Integer>`, each `Integer` object is automatically unboxed:

```
sum = 0;
for (int nextInt : myList) {
    sum += nextInt;
}
```

NOTE: the iterator’s `remove` method cannot be invoked when using the for-each loop syntax.
Each class that implements the `List` interface must provide an `iterator` method

The `Collection` interface extends the `Iterable` interface

All classes that implement the `List` interface (a subinterface of `Collection`) must provide an `iterator` method

Allows use of the Java 5.0 `for-each` loop

```java
public interface Iterable<E> {
    /** returns an iterator over the elements */
    * in this collection. */
    Iterator<E> iterator();
}
```
THE COLLECTIONS FRAMEWORK DESIGN
THE COLLECTION FRAMEWORK

```
<interface> Queue
  <interface> Deque
    AbstractSequentialList
      LinkedList
    Vector
      Stack
  AbstractList
    Vector
    ArrayList

<interface> List

AbstractCollection

<interface> Iterable

<interface> Collection

AbstractSet
  HashSet
    LinkedHashSet
  TreeSet
    ConcurrentSkipListSet
```

Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010
Specifications a subset of methods in the List interface, specifically excluding:
- `add(int, E)`
- `get(int)`
- `remove(int)`
- `set(int, E)`

but including:
- `add(E)`
- `remove(Object)`
- `the iterator method iterator()`
COMMON FEATURES OF COLLECTIONS

- Collections
  + Grow as needed
  + Hold references to objects
  + Have at least two constructors:
    × one to create an empty collection and
    × one to make a copy of another collection
### COMMON FEATURES OF COLLECTIONS (CONT.)

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean add(E obj)</td>
<td>Ensures that the collection contains the object obj. Returns true if the collection was modified.</td>
</tr>
<tr>
<td>boolean contains(E obj)</td>
<td>Returns true if the collection contains the object obj.</td>
</tr>
<tr>
<td>Iterator&lt;E&gt; iterator()</td>
<td>Returns an Iterator to the collection.</td>
</tr>
<tr>
<td>int size()</td>
<td>Returns the size of the collection.</td>
</tr>
</tbody>
</table>
INTRODUCTION

- Consider another part of the Collection hierarchy: the Set interface
- A set is an unordered collection of elements, without duplicates that typically supports efficient membership tests.
  + Elements of a set are like keys of a map, but without any auxiliary values.
- A multiset (also known as a bag) is a set-like container that allows duplicates.
**Sets and the Set Interface**

- **Set objects**
  - Are not indexed
  - Do not reveal the order of insertion of items
  - Enable efficient search and retrieval of information
  - Allow removal of elements without moving other elements around
The union of two sets $A$, $B$ is a set whose elements belong either to $A$ or $B$ or to both $A$ and $B$.
Example: $\{1, 3, 5, 7\} \cup \{2, 3, 4, 5\}$ is $\{1, 2, 3, 4, 5, 7\}$

The intersection of sets $A$, $B$ is the set whose elements belong to both $A$ and $B$.
Example: $\{1, 3, 5, 7\} \cap \{2, 3, 4, 5\}$ is $\{3, 5\}$

The difference of sets $A$, $B$ is the set whose elements belong to $A$ but not to $B$.
Examples: $\{1, 3, 5, 7\} \setminus \{2, 3, 4, 5\}$ is $\{1, 7\}$; $\{2, 3, 4, 5\} \setminus \{1, 3, 5, 7\}$ is $\{2, 4\}$

Set $A$ is a subset of set $B$ if every element of set $A$ is also an element of set $B$.
Example: $\{1, 3, 5, 7\} \subseteq \{1, 2, 3, 4, 5, 7\}$ is true
THE SET INTERFACE AND METHODS

- Required methods:
  + testing set membership,
  + testing for an empty set,
  + determining set size, and
  + creating an iterator over the set

- Optional methods:
  + adding an element (not allow duplicate items ) and
  + removing an element

- Constructors to enforce the “no duplicate members” criterion
  + The add method does not allow duplicate items to be inserted
SET ADT

add(e): Adds the element e to S (if not already present).
remove(e): Removes the element e from S (if it is present).
contains(e): Returns whether e is an element of S.
iterator(): Returns an iterator of the elements of S.

There is also support for the traditional mathematical set operations of union, intersection, and subtraction of two sets S and T:

\[ S \cup T = \{ e: \text{ e is in } S \text{ or e is in } T \}, \]
\[ S \cap T = \{ e: \text{ e is in } S \text{ and e is in } T \}, \]
\[ S - T = \{ e: \text{ e is in } S \text{ and e is not in } T \}. \]

addAll(T): Updates S to also include all elements of set T, effectively replacing S by \( S \cup T \).

retainAll(T): Updates S so that it only keeps those elements that are also elements of set T, effectively replacing S by \( S \cap T \).

removeAll(T): Updates S by removing any of its elements that also occur in set T, effectively replacing S by \( S - T \).
\[ S \cup T = \{ e : e \text{ is in } S \text{ or } e \text{ is in } T \}, \]

```java
setA.addAll(setB);
System.out.println(setA);
Outputs:
[Bill, Jill, Ann, Sally, Bob]
```
\[ S \cap T = \{ e : e \text{ is in } S \text{ and } e \text{ is in } T \}, \]

```java
setACopy.retainAll(setB);
System.out.println(setACopy);
Outputs:
[Jill, Ann]
```
\[ S - T = \{ e : e \text{ is in } S \text{ and } e \text{ is not in } T \}. \]

```
setACopy.removeAll(setB);
System.out.println(setACopy);
Outputs:
[Sally]
```
COMPARISON OF LISTS AND SETS

- Collections implementing the `Set` interface may contain only unique elements.
- Unlike the `List.add` method, the `Set.add` method returns `false` if you attempt to insert a duplicate item.
- Unlike a `List`, a `Set` does not have a `get` method—elements cannot be accessed by index.
You can iterate through all elements in a Set using an Iterator object, but the elements will be accessed in arbitrary order

```java
for (String nextItem : setA) {
    // Do something with nextItem
    ...
}
```
We can implement a set with a list

The space used is $O(n)$