CS–313
Object Oriented Analysis and Design (OOA&D)
Lab # 07

Object-Oriented Programming: Collection classes

Objectives

Understanding the concepts of

- Collection Classes

8.1. The Collection Interface

A Collection represents a group of objects known as its elements. The Collection interface is used to pass around collections of objects where maximum generality is desired. For example, by convention all general-purpose collection implementations have a constructor that takes a `Collection` argument. This constructor, known as a *conversion constructor*, initializes the new collection to contain all of the elements in the specified collection, whatever the given collection's subinterface or implementation type. In other words, it allows you to *convert* the collection's type.

Suppose, for example, that you have a `Collection<String> c`, which may be a `List`, a `Set`, or another kind of `Collection`. This idiom creates a new `ArrayList` (an implementation of the `List` interface), initially containing all the elements in `c`.

```
List<String> list = new ArrayList<String>(c);
```

The following shows the `Collection` interface.

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();
```

```
        // Bulk operations
        boolean containsAll(Collection<?> c);
        // optional
        boolean addAll(Collection<? extends E> c);
        // optional
        boolean removeAll(Collection<?> c);
        // optional
        boolean retainAll(Collection<?> c);
        // optional
        void clear();

        // Array operations
        Object[] toArray();
        <T> T[] toArray(T[] a);
    }
```

The interface does about what you'd expect given that a `Collection` represents a group of objects. The interface has methods to tell you how many elements are in the collection (`size`, `isEmpty`), to check whether a given object is in the collection (`contains`), to add and remove an element from the collection (`add`, `remove`), and to provide an iterator over the collection (`iterator`).

The `add` method is defined generally enough so that it makes sense for collections that allow duplicates as well as those that don't. It guarantees that the `Collection` will contain the specified element after the call completes, and returns `true` if the `Collection` changes as a result of the call. Similarly, the `remove` method is designed to remove a single instance of the specified element from the `Collection`, assuming that it contains the element to start with, and to return `true` if the `Collection` was modified as a result.

Iterators

An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an `Iterator` for a collection by calling its `iterator` method. The following is the `Iterator` interface.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

The `hasNext` method returns `true` if the iteration has more elements, and the `next` method returns the next element in the iteration. The `remove` method removes the last element that was returned by `next` from the underlying `Collection`. The `remove` method may be called only once per call to `next` and throws an exception if this rule is violated.

Note that `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.

Use `Iterator` instead of the `for-each` construct when you need to:

- Remove the current element. The `for-each` construct hides the iterator, so you cannot call `remove`. Therefore, the `for-each` construct is not usable for filtering.
- Iterate over multiple collections in parallel.

1. The Set Interface

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction. The `Set` interface contains *only* methods inherited from `Collection` and adds the restriction that duplicate elements are prohibited. `Set` also adds a stronger contract on the behavior of the `equals` and `hashCode` operations, allowing `Set` instances to be compared meaningfully even if their implementation types differ. Two `Set` instances are equal if they contain the same elements.

The following is the `Set` interface.

```
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

The Java platform contains three general-purpose `Set` implementations: `HashSet`, `TreeSet`, and `LinkedHashSet`. `HashSet`, which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration. `TreeSet`, which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than `HashSet`. `LinkedHashSet`, which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order). `LinkedHashSet` spares its clients from the unspecified, generally chaotic ordering provided by `HashSet` at a cost that is only slightly higher.

2. The List Interface

A `List` is an ordered `Collection` (sometimes called a *sequence*). Lists may contain duplicate elements. In addition to the operations inherited from `Collection`, the `List` interface includes operations for the following:

- `Positional access` — manipulates elements based on their numerical position in the list
- `Search` — searches for a specified object in the list and returns its numerical position
- `Iteration` — extends `Iterator` semantics to take advantage of the list's sequential nature
- `Range-view` — performs arbitrary *range operations* on the list.

The `List` interface follows.

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    // optional
    E set(int index, E element);
    // optional
    boolean add(E element);
    // optional
    void add(int index, E element);
    // optional
    E remove(int index);
    // optional
    boolean addAll(int index, Collection<? extends E> c);

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

The Java platform contains two general-purpose `List` implementations. `ArrayList`, which is usually the better-performing implementation, and `LinkedList` which offers better performance under certain circumstances. Also, `Vector` has been retrofitted to implement `List`.

List Algorithms

Most polymorphic algorithms in the `Collections` class apply specifically to `List`. Having all these algorithms at your disposal makes it very easy to manipulate lists. Here's a summary of these algorithms.

- `sort` — sorts a `List` using a merge sort algorithm, which provides a fast, stable sort. (A *stable sort* is one that does not reorder equal elements.)
- `shuffle` — randomly permutes the elements in a `List`.
- `reverse` — reverses the order of the elements in a `List`.
- `rotate` — rotates all the elements in a `List` by a specified distance.
- `swap` — swaps the elements at specified positions in a `List`.
- `replaceAll` — replaces all occurrences of one specified value with another.
- `fill` — overwrites every element in a `List` with the specified value.
- `copy` — copies the source `List` into the destination `List`.
- `binarySearch` — searches for an element in an ordered `List` using the binary search algorithm.
- `indexOfSubList` — returns the index of the first sublist of one `List` that is equal to another.
- `lastIndexOfSubList` — returns the index of the last sublist of one `List` that is equal to another.

The Map Interface

A `Map` is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. It models the mathematical *function* abstraction. The `Map` interface follows.

```
public interface Map<K,V> {

    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
```

```
    }
}
```

The Java platform contains three general-purpose `Map` implementations: `HashMap`, `TreeMap`, and `LinkedHashMap`. Their behavior and performance are precisely analogous to `HashSet`, `TreeSet`, and `LinkedHashSet`, as described in The Set Interface section. Also, `Hashtable` was retrofitted to implement `Map`.