Java's Collection Framework: Examples

Using `Set`

```
Set set = new HashSet(); // instantiate a concrete set
// ...
set.add(obj); // insert an elements
// ...
int n = set.size(); // get size
// ...
if (set.contains(obj)) {...} // check membership

// iterate through the set
Iterator iter = set.iterator();
while (iter.hasNext()) {
  Object e = iter.next();
  // downcast e
  // ... }
```

Using `Set`

The method call **set1.equals(set2) is true iff**

- **set1 $\subseteq$ set2, and set2 $\subseteq$ set1**

- set1 $\cup$ set2
  - `set1.addAll(set2)`
- set1 $\cap$ set2
  - `set1.retainAll(set2)`
- set1 $-$ set2
  - `set1.removeAll(set2)`

Map

- Basic operations:
  - V put(K ***key***, V ***value***)
    - Returns the previous value associated with key, or null if there was no previous value
  - V get(Object ***key***)
    - Returns null if the key was not found
    - A return value of null *may not* mean the key was not found (some implementations of Map allow null keys and values)

Map

- Tests:
  - boolean containsKey(Object ***key***)
  - boolean containsValue(Object ***value***)
    - *Warning:* probably requires linear time!
  - boolean isEmpty()
  - boolean equals(Object ***o***)
    - Returns true if ***o*** is also a map and has the same mappings

Map

- Optional operations:
  - V put(K ***key***, V ***value***)
    - (So you could implement an immutable map)
  - void putAll(Map ***t***)
    - Adds the mappings from ***t*** to this map
  - void clear()
  - Object remove(Object ***key***)
    - Returns the value that was associated with the key, or null

Map

- Other:
  - int size()
    - Returns the number of key-value mappings
  - int hashCode()
    - Returns a hash code value for this map

Using `Map`

```
Map map = new HashMap(); map.put(key, val);
// insert a key-value pair
// get the value associated with key
Object val = map.get(key);
map.remove(key); // remove a key-value pair
// ...
if (map.containsValue(val)) { ... }
if (map.containsKey(key)) { ... }

Set keys = map.keySet(); // get the set of keys

// iterate through the set of keys
Iterator iter = keys.iterator();
while (iter.hasNext()) {
  Key key = (Key) iter.next();
  // ...}
```

Map views

- Set<K> keySet()
  - Returns a set view of the keys contained in this map.
- Collection<V> values()
  - Returns a collection view of the values contained in this map
  - Can’t be a set—keys must be unique, but values may be repeated

Map views

- Set<Map.Entry<K, V>> entrySet()
  - Returns a set view of the mappings contained in this map.
- A view is *dynamic access* into the Map
  - If you change the Map, the view changes
  - If you change the view, the Map changes
- The Map interface does not provide any Iterators
  - However, there are iterators for the above Sets and Collections

`Map.Entry`: Interface for `entrySet` elements

- ```
  public interface Entry {
      K getKey();
      V getValue();
      V setValue(V value);
  }
  ```
- This is a small interface for working with the Collection returned by `entrySet()`
- Can get elements *only* from the `Iterator`, and they are only valid during the iteration
  ```

Using `TreeMap`

```
TreeMap tm = new TreeMap();
    tm.put("Zara", new Double(3434.34));  …

    Set set = tm.entrySet();  //Map does not implement the Iterator
//entrySet( ) returns a Set containing Map.Entry objects



    Iterator i = set.iterator();
    while(i.hasNext()) {
      Map.Entry me = (Map.Entry)i.next(); //a collection-view of the map
     System.out.print(me.getKey() + ": ");
     System.out.println(me.getValue()); }

    double balance = ((Double)tm.get("Zara")).doubleValue();
    tm.put("Zara", new Double(balance + 1000));
    System.out.println("Zara's new balance: " + tm.get("Zara")); } }
```

Using `Vector`

```
Vector v = new Vector(3, 2);          // initial size is 3, increment is 2
   System.out.println("Initial size: " + v.size());
   System.out.println("Initial capacity: " + v.capacity());

   v.addElement(new Integer(1)); …..
   System.out.println("Capacity after four additions: " + v.capacity());
   v.addElement(new Double(5.45));
   System.out.println("Current capacity: " + v.capacity());
   v.addElement(new Double(6.08));  ….

  System.out.println("First element: " + (Integer)v.firstElement());
  System.out.println("Last element: " + (Integer)v.lastElement());

  if(v.contains(new Integer(3)))
       System.out.println("Vector contains 3.");}
```

Using `ListIterator`

For collections that implement List, you can also obtain an iterator by calling ListIterator which can traverse the list in either direction

```
ArrayList al = new ArrayList();

ListIterator litr = al.listIterator();
while(litr.hasNext()) {
     Object element = litr.next();  …. }

// Now, display the list backwards
System.out.print("Modified list backwards: ");
while(litr.hasPrevious()) {
    Object element = litr.previous();
    System.out.print(element + " "); }
```

Ordering and Sorting

There are two ways to define orders on objects.

- Each class can define a *natural order* among its instances by implementing the `Comparable` interface.

```
int compareTo(Object o)
```

- Arbitrary orders among different objects can be defined by *comparators*, classes that implement the `Comparator` interface.

```
int compare(Object o1, Object o2)
```

This method returns zero if the objects are equal. It returns a positive value if o1 is greater than o2. Otherwise, a negative value is returned.

User-Defined Order

Reverse alphabetical order of strings

```
public class StringComparator
     implements Comparator  {
  public int compare(Object o1, Object o2)
{
    if (o1 != null &&
        o2 != null &&
        o1 instanceof String &&
        o2 instanceof String) {
      String s1 = (String) o1;
      String s2 = (String) o2;
      return - (s1.compareTo(s2));
    } else {
      return 0;
    }
  }
}
```