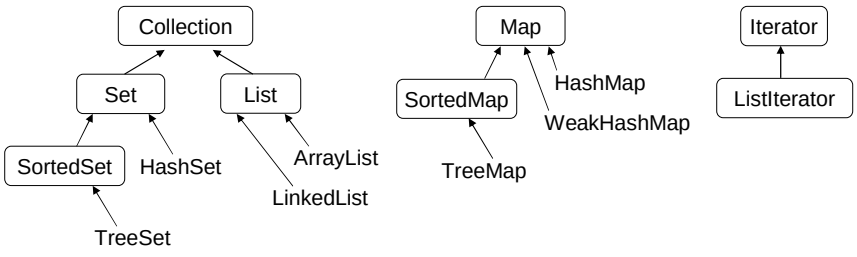
Lecture 6
Collections

Concept

- A collection is a data structure – actually, an object – to hold other objects, which let you store and organize objects in useful ways for efficient access
- Check out the `java.util` package! Lots of interfaces and classes providing a general collection framework. Programmers may also provide implementations specific to their own requirements
- Overview of the interfaces and concrete classes in the *collection framework*



Root interface – Collection (1)

- Methods working with an individual collection
  - `public int size()`
  - `public boolean isEmpty()`
  - `public boolean contains(Object elem)`
  - `public boolean add(Object elem)`
    - Depends on whether the collection allows duplicates
  - `public boolean remove(Object elem)`
  - `public boolean equals(Object o)`
  - `public int hashCode()`
  - `public Iterator iterator()`
  - `public Object[] toArray()`
    - Returns a new array containing references to all the elements of the collection
  - `public Object[] toArray(Object[] dest)`
    - What is returned depends on whether the elements in the collection fit in `dest`
    - If the type of `dest` is not compatible with the types of all elements in the collection, an exception is thrown

Root interface – Collection (2)

- Primary methods operating in bulk from another collection
  - `public boolean containsAll(Collection coll)`
  - `public boolean addAll(Collection coll)`
    - Returns true if any addition succeeds
  - `public boolean removeAll(Collection coll)`
    - Returns true if any removal succeeds
  - `public boolean retainAll(Collection coll)`
    - Removes from the collection all elements that are not elements of `coll`
  - `public void clear()`
    - Remove all elements from this collection
- The SDK does NOT provide any direct implementations of the `Collection` interface
  - Most of the actual collection types implement this interface, usually by implementing an extended interface such as `Set` or `List`
  - This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

Iteration - Iterator

- The `Collection` interface defines an `iterator` method to return an object implementing the `Iterator` interface.
  - It can access the elements in a collection without exposing its internal structure.
  - There are **NO** guarantees concerning the order in which the elements are returned
- Three defined methods in Iterator interface
  - `public boolean` **`hasNext()`** – returns true if the iteration has more elements
  - `public Object` **`next()`** – returns the next element in the iteration
    - An exception will be thrown if there is no next element
    - What's returned is an `Object` object. You may need special casting!
  - `public void` **`remove()`** – remove from the collection the element last returned by the iteration
    - can be called only once per call of `next`, otherwise an exception is thrown

classical routine of using iterator:

```
public void removeLongStrings (Collection coll, int
  maxLen) {
        Iterator it = coll.iterator();
        while ( it.hasNext() ) {
           String str = (String)it.next();
           if (str.length() > maxLen)
           it.remove()
        }
  }
```

Iteration - ListIterator

- `ListerIterator` interface extends `Iterator` interface. It adds methods to manipulate an ordered `List` object during iteration
- Methods
  - `public boolean` **`hasNext()`**`/ public boolean` **`hasPrevious()`**
  - `public Object` **`next()`**`/ public Object` **`previous()`**
  - `public Object` **`nextIndex()`**`/ public Object` **`previousIndex()`**
    - When it's at the end of the list, `nextIndex()` will return `list.size()`
    - When it's at the beginning of the list, `previousIndex()` will return -1
  - public void **`remove()`** – remove the element last returned by `next()` or `previous()`
  - public void **`add(Object o)`**– insert the object o into the list in front of the next element that would be returned by `next()`, or at the end if no next element exists
  - public void **`set(Object o)`** – set the element last returned by `next()` or `previous()` with o

Potential problem of Iterator/ListIterator

- They do NOT provide the *snapshot* guarantee – if the content of the collection is modified when the iterator is in use, it can affect the values returned by the methods

```
import java.util.*;

public class IteratorTest {
  public static void main (String args[]) {
    ArrayList a = new ArrayList();
    a.add("1");
    a.add("2");
    a.add("3");

    Iterator it = a.iterator();
    while(it.hasNext()) {
      String s = (String)(it.next());
      if(s.equals("1")) {
        a.set(2,"changed");
      }

      System.out.println(s);
    }
  }
}
```

Output?

1
2
changed

Potential problem of Iterator/ListIterator (cont.)

- A snapshot will return the elements as they were when the `Iterator/ListIterator` object was created, which is unchangeable in the future
- If you really need a snapshot, you can make a simple copy of the collection
- Many of the iterators defined in the `java.util` package are in the type of *fail-fast* iterators
  - They detect when a collection has been modified
  - When a modification is detected, other than risk performing an action whose behavior may be unsafe, they fail quickly and cleanly by throwing an exception – `ConcurrentModificationException`

```
import java.util.*;

public class IteratorTest2 {
  public static void main (String args[]) {
    ArrayList a = new ArrayList();
    a.add(“1”);
    a.add(“2”);
    a.add(“3”);

    Iterator it = a.iterator();

    a.add(“4”);

    while(it.hasNext()) {
      String s = (String)(it.next());
      System.out.println(s);
    }
  }
}
```

%> javac IteratorTest2.java
%> java IteratorTest2

Exception in thread “main” java.util.ConcurrentModificationException

List

- A `List` is an ordered Collection which allows duplicate elements. Its element indices range from 0 to (`list.size()`-1)
- It adds several methods for an ordered collection
- The interface `List` is implemented by two classes
  1. `ArrayList:` a resizable-array implementation of the `List` interface
     - Adding or removing elements at the end, or getting an element at a specific position is simple – *O(1)*
     - Adding or removing element from the middle is more expensive – *O(n-i)*
     - Can be efficiently scanned by using the indices without creating an `Iterator` object, so it's good for a list which will be scanned frequently
  2. `LinkedList`: a doubly-linked list
     - Getting an element at position *i* is more expensive – *O(i)*
     - A good base for lists where most of the actions are not at the end
- An example of using `LinkedList` (output)

Set and SortedSet

- The `Set` interface provides a more specific contract for its methods, but adding no new methods of its own. A `Set` is a `Collection` that contains ***UNIQUE*** elements.
- The `SortedSet` extends `Set` to specify an additional contract – iterators on such a set will always return the elements in a specified order
  - By default it will be the elements' *natural order* which is determined by the implementation of `Comparable` interface
  - You can specify a `Comparator` object to order the elements instead of the natural order
- There are two implementations of `Set` in the collection framework
  - HashSet – a `Set` implemented using a hashtable
  - TreeSet – a `SortedSet` implemented in a balanced tree structure
- An example of using a `HashSet` (output)

Map and SortedMap

- The `Map` interface does not extend `Collection` interface because a `Map` contains key-value pairs, not only keys. Duplicate keys are not allowed in a `Map`. It's implemented by classes `HashMap` and `TreeMap`.
- There are methods to view the map using collections. For example: `public Set keySet()` and `public Collection values()`.
  - The collections returned by these methods are backed by the Map, so removing an element from one these collections removes the corresponding key/value pair from the map
  - You cannot add elements to these collections
  - If you iterate through the key or value sets, they may return values from their respective sets in any order
- Interface `SortedMap` extends `Map` and maintains its keys in sorted order. Class `TreeMap` implements `SortedMap`.
- An example using `HashMap` (output)

Synchronized wrappers and the `Collections` class (1)

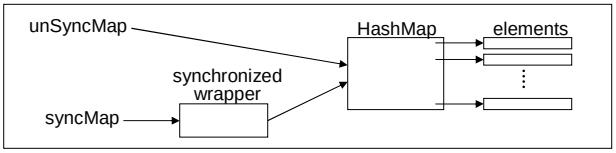
- The `Collections` class contains static utility methods which can be roughly classified into two groups: those provide *wrapped collections* and those don't.
- All the collection implementations provided in `java.util` we've seen so far are unsynchronized
  - concurrent access to a `Collection` by multiple threads could cause indeterminate results or fatal errors.
  - you can use *synchronization wrappers* for those collections that might be accessed by multiple threads to prevent potential threading problems.
- Methods in the `Collections` class to get a synchronized wrapper
  - Collection `synchronizedCollection(Collection c)`
  - Set `synchronizedSet(Set s)`
  - SortedSet `synchronizedSortedSet(SortedSet s)`
  - List `synchronizedList(List l)`
  - Map `synchronizedMap(Map m)`
  - SortedMap `synchronizedSortedMap(SortedMap m)`

Synchronized wrappers and the `Collections` class (2)

- The above methods return wrappers whose methods are fully synchronized, and so are safe to use from multiple threads

Example

```
Map unSyncMap = new HashMap();
Map syncMap = Collections.synchronizedMap(unSyncMap);
```



- `synchMap` has all relevant methods synchronized, passing all calls through to the wrapped map (`unSynchMap`)
- there is actually only one map, but with two different views. So modifications on either map is visible to the other
- the wrapper synchronizes on itself, so you can use `syncMap` to synchronize access, and then use `unsyncMap` safely inside such code

```
synchronized (syncMap) {
    for (int i=0; i< keys.length; i++)
        unSyncMap.put ( keys[i], values[i] );
}
```

Unmodifiable wrappers and the `Collections` class (1)

- The `Collections` class contains a set of methods that return *unmodifiable wrappers* for collections: attempts to modify the returned set, whether direct or via its iterator, result in an `UnsupportedOperationException`
  - The contents of an unmodifiable wrapper can change, but can only through the original collection, not through the wrapper itself
- Six methods to return unmodifable wrappers:
  - `Collection unmodifiableCollection(Collection c)`
  - `Set unmodifiableSet(Set s)`
  - `SortedSet unmodifiableSortedSet(SortedSet s)`
  - `List unmodifiableList(List l)`
  - `Map unmodifiableMap(Map m)`
  - `SortedMap unmodifiableSortedMap(SortedMap m)`

Unmodifiable wrappers and the Collections class (2)

Example

Original: it's dangerous that the array's content can be changed

```
public String suits[]= {
 "Hearts", "Clubs", "Diamonds", "Spades" };
```

Using the unmodifiable wrapper to prevent the danger:

```
private String suitsNames[] = {
 "Hearts", "Clubs", "Diamonds", "Spades" };
public final List suits =
 Collections.unmodifiableList(Arrays.asList(suitNames
);
```

- The unmodifiable wrapper offers *read-only access* to others, while the *read-write access* is still available to the code itself by retaining a reference to the wrapped collection (the original collection)

Abstract implementations

- The collection framework provides a set of abstract implementations for you to design your own implementation of relevant collection interfaces to satisfy your particular needs
- The set of abstract classes:
  - `AbstractCollection`
  - `AbstractSet`
  - `AbstractList`
  - `AbstractSequentialList`
  - `AbstractMap`

The legacy collection types

- The package `java.util` contains some other *legacy collections* than those we just learned. They are still in wide use in existing code and will continue to be used until programmers shift over to the new types
- The set of *legacy collections*
  - `Enumeration` – analogous to `Iterator`
  - `Vector` – analogous to `ArrayList`
  - `Stack` – a subclass of `Vector`
  - `Dictionary` – analogous to `Map` interface
  - `Hashtable` – analogous to `HashMap`
  - `Properties` – a subclass of `HashTable`