Java Collections Framework reloaded

October 1, 2004

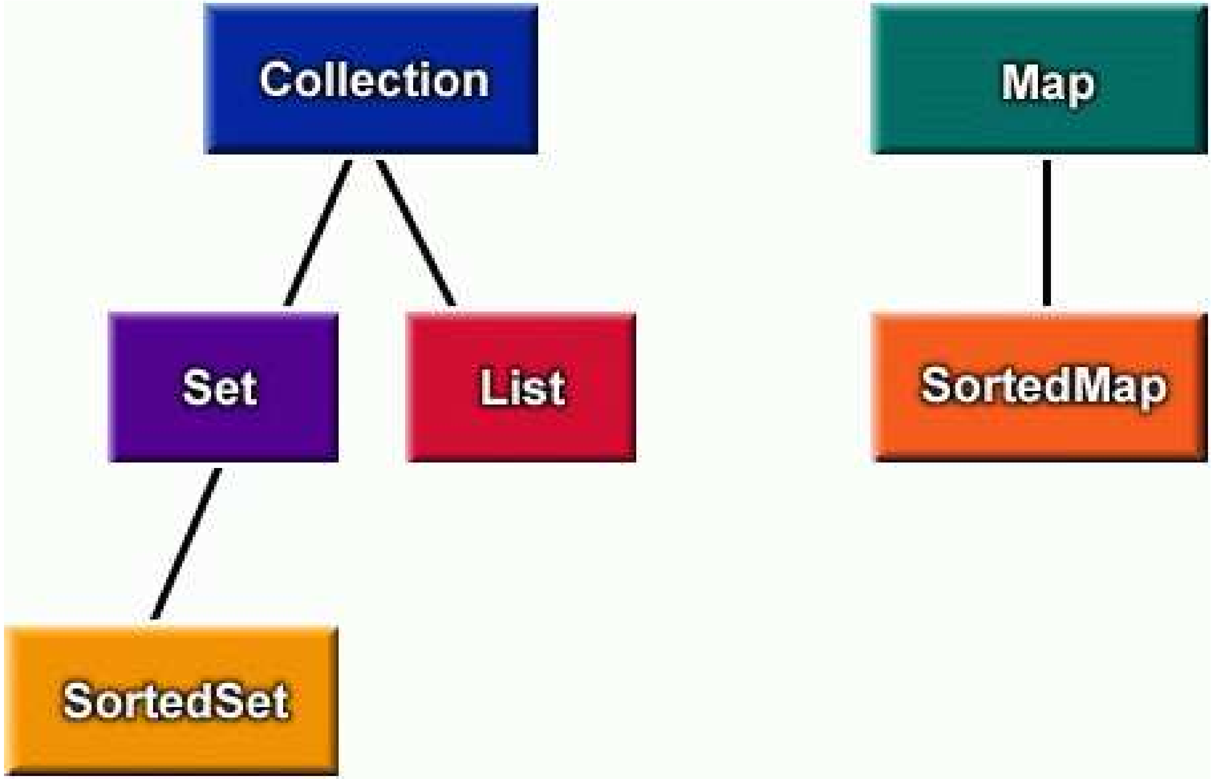
Outline

- Interfaces
- Implementations
- Ordering
- Java 1.5

Components

- Interfaces: abstract data types which allow collections to be manipulated independently of the details of their representation
- Implementations: reusable data structures
- Algorithms: reusable functionality

Core collection interfaces



Core collection interfaces

- Collection
  - represents a group of objects, known as its elements
  - least common denominator that all collections implement
  - is used to pass collections around and manipulate them when maximum generality is desired
- Set
- List
- Map
- SortedSet
- SortedMap

Core collection interfaces

- Collection
- Set
  - collection that cannot contain duplicate elements
- List
  - an ordered collection (sometimes called a sequence)
  - elements can be access by their integer index (position)
- Map
- SortedSet
- SortedMap

Core collection interfaces

- Collection
- Set
- List
- Map
  - an object that maps keys to values
  - cannot contain duplicate keys
  - each key can map to at most one value
- SortedSet
- SortedMap

Core collection interfaces

- Collection
- Set
- List
- Map
- SortedSet
  - a Set that maintains its elements in ascending order
- SortedMap
  - a Map that maintains its mappings in ascending key order

Implementations

	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Set: HashSet, TreeSets and LinkedHashSet

- HashSet
  - constant time for add, $remove$, $contains$ and $size$
  - offers no ordering guarantees
  - iteration is linear in the sum of the number of entries and the number of buckets (the capacity)
- TreeSet
  - implements SortedSet
  - add, $remove$ and $contains$ have $O(\log(n))$ time cost
- LinkedHashSet
  - iteration ordering is the order in which elements were inserted into the set
  - maintains a doubly-linked list running through all of its entries

List: ArrayList and LinkedList

- ArrayList
  - roughly equivalent to Vector, except that it is unsynchronized
  - capacity grows automatically
  - $size$, $isEmpty$, get, set, $iterator$, and $listIterator$ run in constant time
  - add operation runs in amortized constant time (adding n elements requires $O(n)$ time)
- LinkedList
  - provides methods to get, remove and insert an element at the beginning and end of the list (linked lists to be used as a stack, queue, or double-ended queue

Map: HashMap, TreeMap

- HashMap
  - constant-time performance for the basic operations (get and put)
  - iteration requires time proportional to the capacity of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings)
  - when the number of entries in the hash table exceeds the product of the load factor and the current capacity, the capacity is roughly doubled by calling the rehash method
- TreeMap
  - is based on Red-Black tree
  - $O(\log(n))$ time cost for the $containsKey$, get, put and $remove$

Map: LinkedHashMap

- LinkedHashMap
  - iteration ordering is the order in which elements were inserted into the set
  - maintains a doubly-linked list running through all of its entries

Legacy: Vector, Hashtable

- Vector
  - implements List
  - is synchronized
- Hashtable
  - implements Map
  - is synchronized

Wrappers

- Synchronization

```
public static
  Collection
    synchronizedCollection(Collection c);
  Set synchronizedSet(Set s);
  List synchronizedList(List list);
  Map synchronizedMap(Map m);
  SortedSet
    synchronizedSortedSet(SortedSet s);
  SortedMap
     synchronizedSortedMap(SortedMap m);
```

- Unmodifiable

Wrappers

- Synchronization
- Unmodifiable

```
public static
  Collection
    unmodifiableCollection(Collection c);
  Set unmodifiableSet(Set s);
  List unmodifiableList(List list);
  Map unmodifiableMap(Map m);
  SortedSet
    unmodifiableSortedSet(SortedSet s);
  SortedMap
    unmodifiableSortedMap(SortedMap m);
```

Special Implementations

- List-view of an Array

```
List l = Arrays.asList(new Object[size]);
```

- Immutable Multiple-Copy List

```
List l = new ArrayList(
           Collections.nCopies(1000, null));
lovablePets.addAll(
      Collections.nCopies(69, "fruit bat"));
```

- Immutable Singleton Set

```
c.removeAll(Collections.singleton(e));
```

- Empty Set and Empty List Constants

```
static Set Collections.EMPTY_SET;
statis List Collections.EMPTY_LIST;
```

Algorithms

- are implemented in Collections
- for List:
  - sorting: uses a slightly optimized merge sort algorithm
  - shuffling
  - reverse, fill, copy
  - searching: binarySearch
- any Collection:
  - finding extreme values: min, max

More interfaces

- Comparator

  ```
  int compare(Object o1, Object o2)
  boolean equals(Object obj)
  ```

- Comparable

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

  The natural ordering for a class `C` is said to be consistent with equals if and only if `(e1.compareTo((Object)e2) == 0)` has the same boolean value as `e1.equals((Object)e2)` for every `e1` and `e2` of class `C`.

Object

- `public boolean equals(Object obj)`
- `public int hashCode()`

If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.

Java 1.5

- new things:
    - Generics
    - for-each

```
void cancelAll(Collection<TimerTask> c) {
  for (Iterator<TimerTask> i = c.iterator();
                                i.hasNext(); )
    i.next().cancel();
}

void cancelAll(Collection<TimerTask> c) {
  for (TimerTask t : c)
    t.cancel();
}
```

Java 1.5

```
// Returns the sum of the elements of a
int sum(int[] a) {
    int result = 0;
    for (int i : a)
        result += i;
    return result;
}
```

That’s all!

Have a nice weekend!