# Utility or Collection Framework

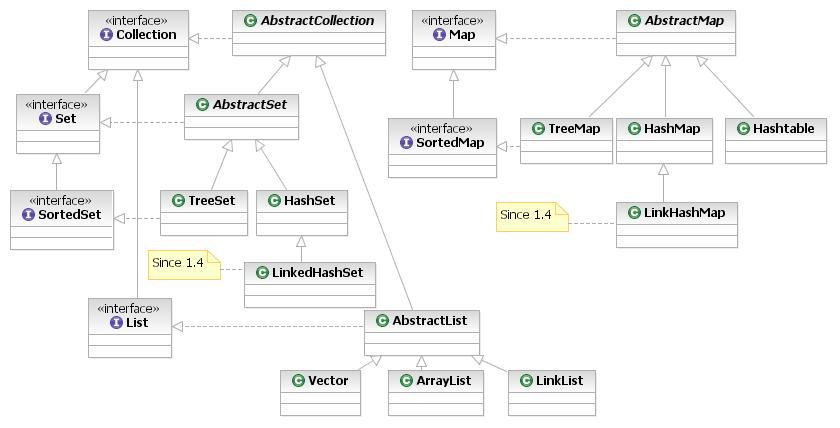
**Why Collection Framework**

Collection are nothing but group of objects stored in well defined manner. **Earlier, Arrays are used to represent these groups of objects. But, arrays are not re-sizable. Size of the arrays is fixed.** Size of the arrays can not be changed once they are defined. This causes lots of problem while handling group of objects. To overcome this drawback of arrays, Collection framework or simply collections are introduced in java from JDK 1.2.

Although, there were classes like **Dictionary, Vector, Stack and Properties** which handle group of objects better than the arrays. **But, each of them handles the objects differently**. The way you use Dictionary class is totally different from the way you use Stack class and the way you use Vector class is different from the way you use Properties class. Hence, there needed a central and unifying theme to handle the group of objects. That’s why we need collection framework.

**What is Collection Framework?**

Collection Framework in java is a **centralized and unified theme to store and manipulate the group of objects**. Java Collection Framework provides some pre-defined classes and interfaces to handle the group of objects. Using collection framework, you can store the objects **as a list or as a set or as a queue or as a map and perform operations like adding an object or removing an object or sorting the objects** without much hard work.

**Class Hierarchy of Collection Framework**

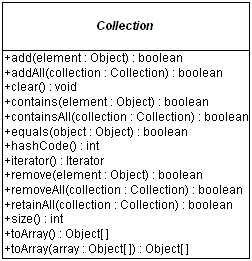
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Interface** | **Implementation** | | | | **Historical** |  |
| Set | HashSet |  | TreeSet | LinkedHashSet |  | It handles list of objects which must contain unique element. |
| List |  | ArrayList |  | LinkedList | Vector Stack | It handles sequential list of objects. |
| Map | HashMap |  | TreeMap |  | Hashtable Properties | This is the one interface in Collection Framework which is not inherited from Collection interface. It handles group of objects as Key/Value pairs |
| Queue |  | PriorityQueue |  | LinkedList |  | It handles special list of objects in which elements are removed only from the head. |

The entire collection framework is divided into four interfaces.

Collection: A *collection* is an object that represents a group of objects (such as Vector, ArrayList, etc). The primary advantages of a collections framework are that it:  
 i) **Reduces programming effort** by providing useful data structures and algorithms so you don't have to   
 write them yourself.  
 ii) **Increases performance** by providing high-performance implementations of useful data structures and   
 algorithms. Because the various implementations of each interface are interchangeable, programs can be   
 easily tuned by switching implementations.  
 iii) **Reduces the effort required to learn APIs** iv) **Reduces the effort required to design and implement APIs**

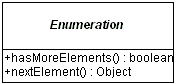
### Collection Interface

There are fourteen *collection interfaces*. The most basic interface is ***Collection***. These interfaces extend Collection: Set, List, SortedSet, NavigableSet, Queue, Deque, BlockingQueue and BlockingDeque. The other collection interfaces, Map, SortedMap, NavigableMap, ConcurrentMap and ConcurrentNavigableMap do not extend Collection, as they represent mappings rather than true collections. However, these interfaces contain *collection-view* operations, which allow them to be manipulated as collections.



**Enumeration Interface**

A class that implements Enumeration interface provides access to series of elements one at a time. You need to call nextElement method to get next element in the series. Also hasMoreElements () method gives you status about the availability of next element in the series.



**Example -** com.src.java.collection – MyEnumeration.java

**Question: Why Java1.5 onwards Java has retricted to store Homogenous type of Objects inside any Collection using Generics? OR Why java has introduced Generic Java1.5 onwards?**

**Iterator Interface**  
All of the collection classes provides iterator () method to iterate through the collection. The iterator () method returns the Iterator object through which you can access the collection elements in an order. Enumeration also does the same purpose. The difference between Iterator and Enumerations is: **Iterators allow the caller to remove elements from the underlying collection** during the iteration with well-defined semantics.



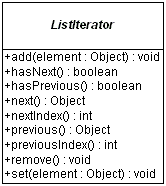
**Example -** com.src.java.collection – MyCollectionIterator.java, MyItrRemoveElement.java

**Question: Difference between Enumeration & Iterator?**

Only major difference between Enumeration and iterator is Iterator has a remove () method while Enumeration doesn't. Enumeration acts as Read-only interface, because it has the methods only to traverse and fetch the objects, where as by using Iterator we can manipulate the objects like adding and removing the objects from collection e.g. Arraylist.

Also Iterator is more secure and safe as compared to Enumeration because it does not allow other thread to modify the collection object while some thread is iterating over it and throws **ConcurrentModificationException**.  
  
**ListIterator Interface**

The *ListIterator* interface extends the *Iterator* interface to support **bi-directional access**, as well as adding or changing elements in the underlying collection. Using ListIterator, we can iterate all elements of a list in either direction. You can access next element by calling next () method, and also you can access previous element by calling previous () method on the list.



**Example -** com.src.java.collection – MyListIterator.java

**Question: Difference between Iterator & ListIterartor?**

|  |  |
| --- | --- |
| **Iterator** | **ListIterator** |
| Iterator () method is available for all collections. That is, Iterator can be used for all collection classes. | ListIterator () method is available for those collections that implement List interface. That is, descendents of List interface only can use ListIterator. |
| Can do remove operation only on elements | Can remove, add and replace elements |

**How to use Iterator**

There are two kinds of Iterator in Java, **fail-safe (ConcurrentHashMap)** and**fail-fast (ArrayList, HashMap, HashSet, etc)**. Fail-safe Iterator doesn't throw **ConcurrentModificationException** during iteration while fail-fast does, if, Iterator realizes any structural change in Collection once Iteration begins.

**Example -** com.src.java.collection – IteratorHowToUse.java

### Group Operations

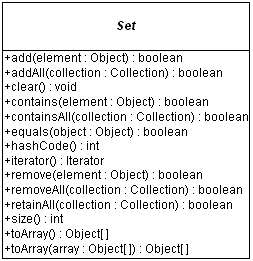
Other operations the Collection interface supports are tasks done on groups of elements or the entire collection at once:

* boolean containsAll(Collection collection)
* boolean addAll(Collection collection)
* void clear()
* void removeAll(Collection collection)
* **void retainAll(Collection collection)**

The containsAll() method allows you to discover if the current collection contains all the elements of another collection, a subset. The remaining methods are optional, in that a specific collection might not support the altering of the collection. The addAll() method ensures all elements from another collection are added to the current collection, usually a union. The clear() method removes all elements from the current collection. The removeAll() method is like clear() but only removes a subset of elements. The retainAll() method is similar to the removeAll() method, but does what might be perceived as the opposite. It removes from the current collection those elements not in the other collection, an intersection.

### Set Interface

The Set interface extends the Collection interface and, by definition, forbids duplicates within the collection. All the original methods are present and no new methods are introduced. The concrete Set implementation classes rely on the equals() method of the object added to check for equality.



### HashSet, TreeSet, LinkedHashSet

**HastSet**

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. **This class permits the null element**.

If we are trying to insert duplicate objects we won't get compile time error and runtime error add() method simply returns false.

**Note that this implementation is not synchronized.** If multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it must be synchronized externally.

Set s = Collections.synchronizedSet(new HashSet(...));

The iterators returned by this class's iterator method are **fail-fast**: if the set is modified at any time after the iterator is created, in any way except through the iterator's own removemethod, the Iterator throws a [ConcurrentModificationException](http://java.util).

**Example -** com.src.java.collection – SpellChecker.java

**TreeSet**It stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet.  
The TreeSet implementation is useful when you need to extract elements from a collection in a sorted manner. In order to work property, elements added to a TreeSet must be sortable. The Collections Framework adds support for Comparable elements. It is generally faster to add elements to a HashSet, and then convert the collection to a TreeSet for sorted traversal.  
To optimize HashSet space usage, you can tune the initial capacity and load factor. The TreeSet has no tuning options.

**Example -** com.src.java.collection – TreeSetNaturalOrder.java

**LinkedHashSet**

It 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).

**Example -** com.src.java.collection – SetComparision.java

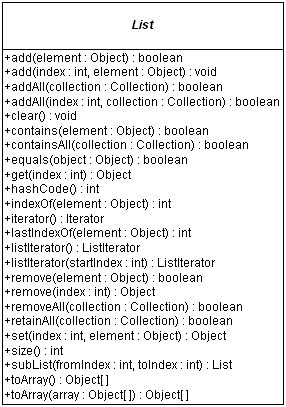
**Question: Difference among HashSet, TreeSet & LinkedHashSet?**

HashSet is fastest, LinkedHashSet is second on performance or almost similar to HashSet but TreeSet is bit slower because of sorting operation it needs to perform on each insertion.  
HashSet is backed by a **HashMap instance**, LinkedHashSet is implemented using **HashSet and LinkedList** while TreeSet is backed up by **NavigableMap** in Java and by default it uses **TreeMap**.

HashSet and LinkedHashSet allow null but TreeSet doesn't allow null but TreeSet doesn't allow null and throw [java.lang.NullPointerException](http://javarevisited.blogspot.sg/2012/06/common-cause-of-javalangnullpointerexce.html) when you will insert null into TreeSet. Since TreeSet uses [compareTo () method](http://javarevisited.blogspot.sg/2011/11/how-to-override-compareto-method-in.html) of respective elements to compare them which throws NullPointerException while comparing with null.

### List Interface

The List interface extends the Collection interface to define an ordered collection, permitting duplicates.



### Vector, ArrayList, LinkedList Classes

**Vector**

Vectors (the java.util.Vector class) are commonly used instead of arrays, because they expand automatically when new data is added to them. The Java 2 Collections API introduced the similar ArrayList data structure. ArrayLists are unsynchronized and therefore faster than Vectors, but less secure in a multithreaded environment. The Vector class was changed in Java 2 to add the additional methods supported by ArrayList.

**Example -** com.src.java.collection – VectorExample.java

**ArrayList**

It is resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. This class is roughly equivalent to Vector, except that it is unsynchronized.

If you need to support random access, without inserting or removing elements from any place other than the end, than ArrayList offers the optimal collection.

**Note that this implementation is not synchronized.** If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally.

*List list = Collections.synchronizedList(new ArrayList(...));*

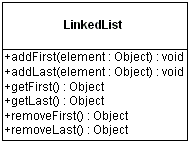
The iterators returned by this class's [iterator](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html#iterator()) and [listIterator](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html#listIterator(int)) methods are *fail-fast*: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own [remove](http://docs.oracle.com/javase/7/docs/api/java/util/ListIterator.html#remove()) or [add](http://docs.oracle.com/javase/7/docs/api/java/util/ListIterator.html#add(E)) methods, the iterator will throw a [ConcurrentModificationException](http://java.util).

**Example -** com.src.java.collection – SwapArrayList.java

**LinkedList**

If, however, you need to frequently add and remove elements from the middle of the list and only access the list elements sequentially then LinkedList offers the better implementation. Both ArrayList and LinkedList implement the Cloneable interface. In addition, LinkedList adds several methods for working with the elements at the ends of the list

Example – Mail Box Application always prefer to use LinkList



**Example -** com.src.java.collection – Node.java, SinglyLinkedListImpl.java, SingleLinkedListClient.java

**Question- What is difference between Collection & Collections?**

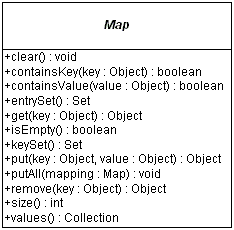
**Question- How to sort List of Employees based on salary?  
Example -** com.src.java.collection – Employee.java, SortEmployees.java

**Question- What is difference between Array, Vector & ArrayList?**Array is fixed size length data structure whereas Vector & ArrayList are auto expandable. ArrayList is not synchronised which will give better performance whereas Vector preferable to used in Multithreaded environment since it is synchronized.

**Question- What is difference between ArrayList & LinkedList?**ArrayList is used for fast searching based on Index whereas LinkedList we can use where lots of insertion & deletion operations are required. ArrayList is slower for insertion & deletion since array element shift their position upto last element which takes more time in comparison to LinkedList.

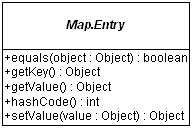
### Map Interface

The Map interface is not an extension of the Collection interface. Instead, the interface starts off its own interface hierarchy, for maintaining key-value associations. The interface describes a mapping from keys to values, without duplicate keys, by definition.



### Map.Entry Interface

The entrySet () method of Map returns a collection of objects that implement Map.Entry interface. Each object in the collection is a specific key-value pair in the underlying Map.



### HashMap, TreeMap Classes

For inserting, deleting, and locating elements in a Map, the HashMap offers the best alternative. If, however, you need to traverse the keys in a sorted order, then TreeMap is your better alternative. Depending upon the size of your collection, it may be faster to add elements to a HashMap, and then convert the map to a TreeMap for sorted key traversal. Using a HashMap requires that the class of key added have a well-defined hashCode () implementation. With the TreeMap implementation, elements added to the map must be sortable.

To optimize HashMap space usage, you can tune the initial capacity and load factor. The TreeMap has no tuning options, as the tree is always balanced.

The Hashtable and Properties classes are historical implementations of the Map interface.

**HashMap**

**Example -** com.src.java.collection – HashMapDemo.java, Account.java, TestMutableKey.java

**Question- How to sort a Map either based on key / value?  
Example -** com.src.java.collection – SortMapByKeyAndValue.java

### Performance comparison of different ways to iterate over HashMap

To compare traversal in hash map in java. HashMap is very frequently used class and most of the times we just fetch the value using get(Object key) method provided by class. But at times it is required to iterate over whole Map and fetch all key-value pairs stored in it. For example, analyzing all request parameters sent from client. If you are using this for every client you will be iterating whole map at least once in your code.

If you are using this type of iteration many places in code and there are large number of requests then you surely would like to optimize your iteration code to make best use of it.

Now let’s compare their performances for a common data set stored in map. I am storing 10 lacs key value pairs in map and will iterate over map in all four ways. I will also fetch key and value from map for all 10 lacs entries in best suitable way. Then i will capture the time taken by each way.

**Observations-** 10 lacs is very big number of most the application requirements. Even though the **difference is not very substantial in milliseconds**, as compare to it was [very big in case of for loops](http://java). I believe most of us can live with such a minor difference.

But if you want to be very specifically make the conclusion, using entry set is more powerful and yields better performance as comparing to using key set for iteration. **Result varies from 20% -- 50% when above program is executed multiple times.**

**Example -** com.src.java.collection – DifferentWaysToIterateOverHashMap.java

**TreeMap**

**Example -** com.src.java.collection – MyTreeMapComparator.java, SortMapByKeyAndValue.java

### WeakHashMap Class

A hashtable-based Map implementation with weak keys. An entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use. More precisely, the presence of a mapping for a given key will not prevent the key from being discarded by the garbage collector, which is, made finalizable, finalized, and then reclaimed. When a key has been discarded its entry is effectively removed from the map, so this class behaves somewhat differently than other Map implementations. Both null values and the null key are supported.

Like most collection classes, this class is not synchronized. A synchronized WeakHashMap may be constructed using the Collections.synchronizedMap method.

**Example -** com.src.java.collection – WeakHashMapExample.java

**Question- Where can we use WeakHashMap?**   
WeakHashMap to avoid memory leaks. WeakHashMap can also be used fairly straightforwardly as a cache that can be cleared automatically when JVM memory is low.

### ConcurrentHashMap Class

The ConcurrentHashMap is very similar to the java.util.HashTable class, except that ConcurrentHashMap offers better concurrency than HashTable does. ConcurrentHashMap does not lock the Map while you are reading from it. Additionally, ConcurrentHashMap does not lock the entire Map when writing to it. It only locks the part of theMap that is being written to, internally.

Another difference is that ConcurrentHashMap does not throw ConcurrentModificationException if theConcurrentHashMap is changed while being iterated. The Iterator is not designed to be used by more than one thread though.

**Example -** com.src.java.collection – ConcurrentHashMapExample.java