Collection

Framework

By
Apparao G

Topics:

What is and Why Collections?

Core Collection Interfaces

Implementations

Algorithms

Custom Implementations

What is a Collection?

Collection is an object that groups multiple elements into a single unit.

Collections are used to store, retrieve, manipulate, and communicate aggregate data.

Some collections allow duplicate elements and others do not. Some are ordered and others unordered.

All collections frameworks contain the following:

1.Interfaces

2.Implementations

3.Algorithms

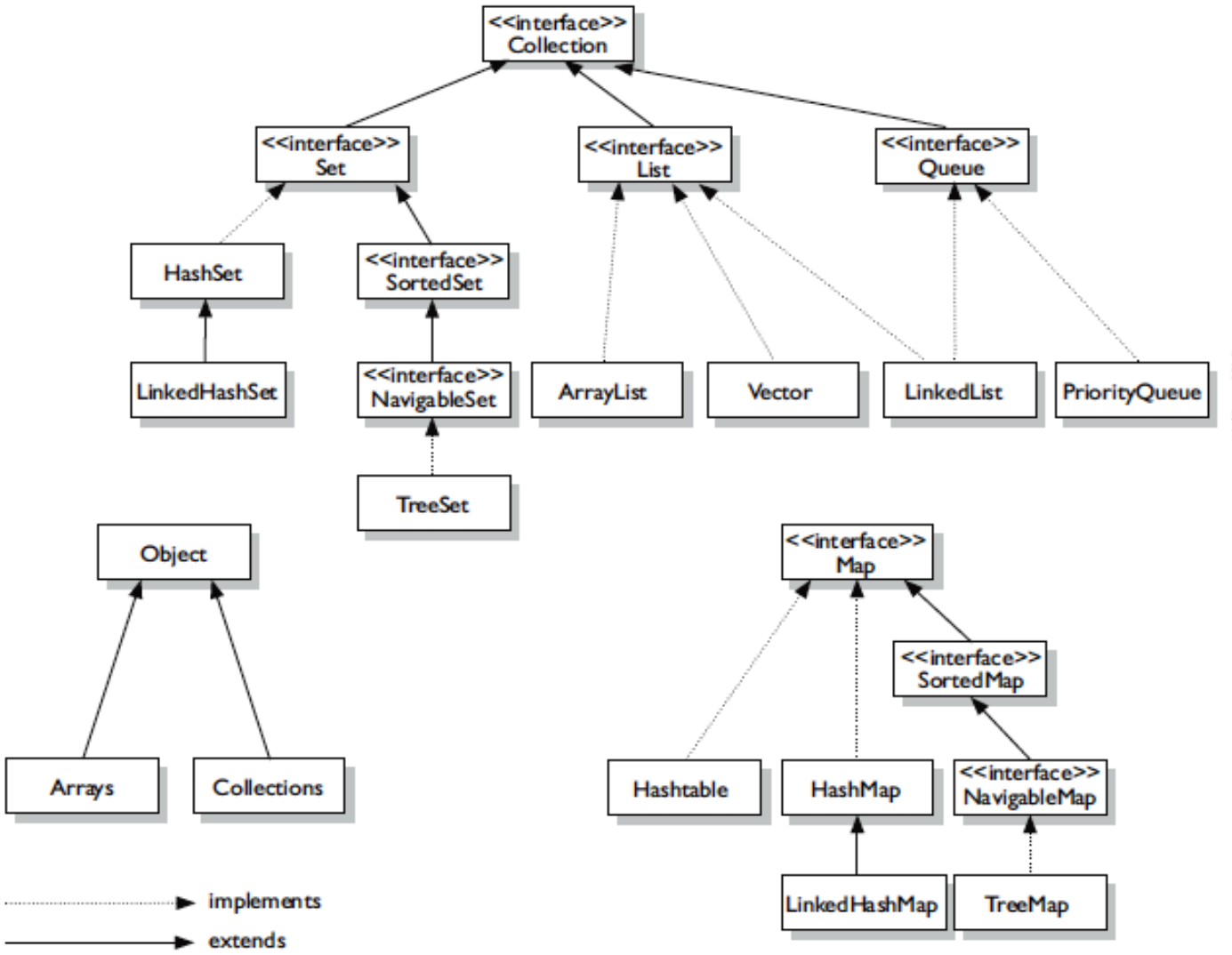
Benefits of Collection Framework:

- Reduces programming effort
- Increases program speed and quality
- Allows interoperability among unrelated APIs
  > The collection interfaces are the vernacular by which APIs
- pass collections back and forth
- Reduce effort to learn and use new APIs
- Reduces effort to design new APIs
- Fosters software reuse
- interfaces are by nature reusable

Interfaces:

- Collection interfaces are abstract data types that represent collections.
- Collection interfaces are in the form of Java interfaces
- Interfaces allow collections to be manipulated independently of the implementation details of their representation
- Polymorphic behavior
- In Java programming language (and other object oriented languages), interfaces generally form a hierarchy
- You choose one that meets your need as a type

Hierarchy of Collection Framework:



Methods of Collection interface :

Method	Description
public boolean add(Object element)	used to insert an element in this collection.
public boolean addAll(collection c)	used to insert the specified collection elements in the invoking collection.
public boolean remove(Object element)	used to delete an element from this collection.
public boolean removeAll(Collection c)	used to delete all the elements of specified collection from the invoking collection.
public boolean retainAll(Collection c)	used to delete all the elements of invoking collection except the specified collection.
public int size()	return the total number of elements in the collection.
public void clear()	removes the total no of element from the collection.
public boolean contains(object element)	used to search an element.
public boolean containsAll(Collection c)	used to search the specified collection in this collection.
public Iterator iterator()	returns an iterator.
public Object[] toArray()	converts collection into array.
public boolean isEmpty()	checks if collection is empty.

ArrayList:

1.ArrayList class maintains the insertion order
2. it is non-synchronized.

3.The elements stored in the ArrayList class can be randomly accessed.

4. ArrayList increases by half of its size when the number of elements exceeds from its capacity.

5. It uses dynamically resized arrays.

6. Default capacity of ArrayList is 10.

Example:

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

ArrayList<String> list = new ArrayList<String>(20);
List<String> list = new ArrayList<String>();
```

Iterator interface :

- Iterator interface provides the facility of iterating the elements in forward direction only.

Methods of Iterator interface

- There are only three methods in the Iterator interface. They are mentioned below.

public boolean hasNext() it returns true if iterator has more elements.

public object next() it returns the element and moves the cursor pointer to the next element.

public void remove() it removes the last elements returned by the iterator. It is rarely used.

- **Collection Sorting:**
  We can sort the collection contains following elements using Collections.sort() method.
  1. String objects
  2. Wrapper class objects
  3. User-defined class objects

```
public class StringSort{
public static void main(String args[]){
        ArrayList<String> al=new ArrayList<String>();
        al.add("Vizag");
        al.add("Hyderabad");
        al.add("Mumbai");
        al.add("Bangalore");
        Collections.sort(al);
        Iterator itr=al.iterator();
        while(itr.hasNext()){
                System.out.println(itr.next());
        }
   }
}
```

Sort string objects in reverse order:

```
class StringSortReverse{
public static void main(String args[]){
        ArrayList<String> al=new ArrayList<String>();
        al.add("Vizag");
        al.add("Hyderabad");
        al.add("Mumbai");
        al.add("Bangalore");
        Collections.sort(al,Collections.reverseOrder());
        Iterator itr=al.iterator();
        while(itr.hasNext()){
                System.out.println(itr.next());
        }
    }
}
```

Sort Wrapper class objects:

```
public class WrapperObjectsSort{
public static void main(String args[]){
        ArrayList<String> al=new ArrayList<String>();
        al.add(Integer.valueOf(101));
        al.add(Integer.valueOf(30));
        al.add(Integer.valueOf(400));
        al.add(Integer.valueOf(200));
        Collections.sort(al,Collections.reverseOrder());
        Iterator itr=al.iterator();
        while(itr.hasNext()){
                System.out.println(itr.next());
        }
    }
}
```

- **Comparable:**
  Comparable interface is used to order/sort the objects of the user-defined class.
  It contains only one method named compareTo(Object).
  It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only.

public int compareTo(Object obj): It is used to compare the current object with the specified object. It returns

- positive integer, if the current object is greater than the specified object.
- negative integer, if the current object is less than the specified object.
- zero, if the current object is equal to the specified object.

```
public class Employee implements Comparable<Employee>{
		private int eno;
		private String name;
		private int sal;
		public Employee (int eno,String name,int sal){
				this.eno=eno;
				this.name=name;
				this.sal=sal;
		}

		public int compareTo(Employee  emp){
				if(sal==emp.sal)
						return 0;
				else if(sal>emp.sal)
						return 1;
				else
						return -1;
		}
}
```

```
public class ComparableTest{
        public static void main(String args[]){
                ArrayList<Employee> al=new ArrayList<Employee>();
                al.add(new Employee(100,"Ramu",2000));
                al.add(new Employee(200,"Balu",1000));
                al.add(new Employee(300,"Amish",1500));
                Collections.sort(al);
                for(Employee emp:al){
                        System.out.println(emp.eno+" "+emp.name+" "+emp.sal);
                }
        }
}
```
output:

Reverse order:

```
public class Employee implements Comparable<Employee>{
		private int eno;
		private String name;
		private int sal;
		public Employee (int eno,String name,int sal){
				this.eno=eno;
				this.name=name;
				this.sal=sal;
		}

		public int compareTo(Employee  emp){
				if(sal==emp.sal)
						return 0;
				else if(sal<emp.sal)
						return 1;
				else
						return -1;
		}
}
```

```
public class ComparableTest{
	public static void main(String args[]){
		ArrayList<Employee> al=new ArrayList<Employee>();
		al.add(new Employee(100,"Ramu",2000));
		al.add(new Employee(200,"Balu",1000));
		al.add(new Employee(300,"Amish",1500));
		Collections.sort(al);
		for(Employee emp:al){
			System.out.println(emp.eno+" "+emp.name+"
"+emp.sal);
		}
	}
}
```

Comparator:

1. It is used to sort the objects of a user-defined class.

2. It provides multiple sorting sequences.

3. It uses compare() method to compare the objects.

public int compare(Object obj1, Object obj2)

It compares the first object with the second object.

Compare() method returns zero if the objects are equal.

It returns a positive value if obj1 is greater than obj2. Otherwise, a negative value is returned.

```
public class Employee {
                private int eno;
                private String name;
                private int sal;
                public Employee (int eno,String name,int sal){
                                this.eno=eno;
                                this.name=name;
                                this.sal=sal;
                }
  }
Public class SalaryComparator implements Comparator{
                public int compare(Object o1,Object o2){
                Employee e1=(Employee)o1;
                Employee e2=(Employee)o2;

                if(e1.sal==e2.sal)
                                return 0;
                else if(e1.sal>e2.sal)
                                return 1;
                else
                                return -1;
                }
}
```

```
class NameComparator implements Comparator{
		public int compare(Object o1,Object o2){
				Employee e1=(Employee)o1;
				Employee e2=(Employee)o2;
				return e1.name.compareTo(e2.name);
		}
}
```

Sort Based on Name:

```
public class NameComparatorTest{
		public static void main(String args[]){
				ArrayList<Employee> al=new ArrayList<Employee>();
				al.add(new Employee(100,"Ramu",2000));
				al.add(new Employee(200,"Balu",1000));
				al.add(new Employee(300,"Amish",1500));
				Collections.sort(al,new NameComparator());
				for(Employee emp:al){
					System.out.println(emp.eno+" "+emp.name+" "+emp.sal);
				}
		}
}
```

```
public class SalaryComparatorTest{
        public static void main(String args[]){
                ArrayList<Employee> al=new ArrayList<Employee>();
                al.add(new Employee(100,"Ramu",2000));
                al.add(new Employee(200,"Balu",1000));
                al.add(new Employee(300,"Amish",1500));
                Collections.sort(al,new SalaryComparator());
                for(Employee emp:al){
                        System.out.println(emp.eno+" "+emp.name+" "+emp.sal);
                }
        }
}
```

unmodifiableList (Immutable List):

Returns an unmodifiable view of the specified list. This method allows modules

to provide users with "read-only" access to internal lists.

Query operations on the returned list "read through" to the specified list, and attempts

to modify the returned list, whether direct or via its iterator, result in an UnsupportedOperationException.

Signature:

public static <T> List<T> unmodifiableList(List<? extends T> list)

Parameters:

list - the list for which an unmodifiable view is to be returned.

Returns:

an unmodifiable view of the specified list.

Example:

Collections.unmodifiableList(list);

synchronizedList:

Returns a synchronized (thread-safe) list backed by the specified list.In order to guarantee serial access, it is critical that all access to the backing list is accomplished through the returned list.

Signature:

public static <T> List<T> synchronizedList(List<T> list)

It is imperative that the user manually synchronize on the returned list when iterating over it:

```
List list = Collections.synchronizedList(new ArrayList());
    ...
 synchronized (list) {
    Iterator i = list.iterator(); // Must be in synchronized block
    while (i.hasNext())
       foo(i.next());
}
```

Vector:

1.Vector is synchronized.
2.Vector increments 100% means doubles the array size if the total number of elements exceeds than its capacity.
3.Vector is slow because it is synchronized
4.A Vector can use the Iterator interface or Enumeration interface to traverse the elements.

```
class VectorTest{
 public static void main(String args[]){
  Vector<String> v=new Vector<String>();
  v.add("Bangalore");
  v.addElement("Hyderabad");
  Enumeration e=v.elements();
  while(e.hasMoreElements()){
   System.out.println(e.nextElement());
  }
 }
}
```

LinkedList:

1. LinkedList internally uses a doubly linked list to store the elements.
2. Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3. It maintains the insertion order and is not synchronized.
4. LinkedList is better for manipulating data.
5. LinkedList class can be used as a list, stack or queue.

```
public class LinkedListTest{
public static void main(String args[]){
        LinkedList<String> ll=new LinkedList<String>();
        ll.add(“India");
        ll.add(“USA");
        Iterator<String> itr=ll.iterator();
        while(itr.hasNext()){
                System.out.println(itr.next());
        }
    }
}
```

Map Interface:

- A map contains values on the basis of key, i.e. key and value pair.
- Each key and value pair is known as an entry.
- A Map contains unique keys.
- A Map is useful if you have to search, update or delete elements on the basis of a key.
- Implementation classes are HashMap, LinkedHashMap, and TreeMap.
- Map doesn't allow duplicate keys, but it can have duplicate values.
- A Map can't be traversed, so you need to convert it into Set using *keySet()* or *entrySet()* method.

Method	Description
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.
V putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.
V remove(Object key)	It is used to delete an entry for the specified key.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.
Set keySet()	It returns the Set view containing all the keys.
Set<Map.Entry<K,V>> entrySet()	It returns the Set view containing all the keys and values.
void clear()	It is used to reset the map.

boolean containsValue(Object value)	This method returns true if some value equal to the value exists within the map, else return false.
boolean containsKey(Object key)	This method returns true if some key equal to the key exists within the map, else return false.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
V get(Object key)	This method returns the object that contains the value associated with the key.
V getOrDefault(Object key, V defaultValue)	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
int hashCode()	It returns the hash code value for the Map
boolean isEmpty()	This method returns true if the map is empty; returns false if it contains at least one key.
V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V replace(K key, V value)	It replaces the specified value for a specified key.
boolean replace(K key, V oldValue, V newValue)	It replaces the old value with the new value for a specified key.
void replaceAll(BiFunction<? super K,? super V,? extends V> function)	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
Collection values()	It returns a collection view of the values contained in the map.

HashMap:

1.HashMap is the implementation of Map, but it doesn't maintain any order.

2.HashMap allows null keys and values

3.HashMap class is non synchronized.

4.The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

5.HashMap contains only unique keys.

HashMap(): It is used to construct a default HashMap.

HashMap(Map<? extends K,? extends V> m): It is used to initialize the hash map by using the elements of the given Map object m.

HashMap(int capacity): It is used to initializes the capacity of the hash map to the given integer value, capacity.

HashMap(int capacity, float loadFactor):It is used to initialize both the capacity and load factor of the hash map by using its arguments.

How to make Map as immutable?

public static <K,V> Map<K,V> unmodifiableMap(Map<? extends K,? extends V> m)

Returns an unmodifiable view of the specified map. This method allows modules to provide users with "read-only" access to internal maps. Query operations on the returned map "read through" to the specified map, and attempts to modify the returned map, whether direct or via its collection views, result in
an UnsupportedOperationException.

Parameters: m - the map for which an unmodifiable view is to be returned.
Returns: an unmodifiable view of the specified map.

How to synchronize the Map?

Signature:

public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)

Returns a synchronized (thread-safe) map backed by the specified map. In order to guarantee serial access, it is critical that all access to the backing map is accomplished through the returned map.

It is imperative that the user manually synchronize on the returned map when iterating over any of its collection views:

```
Map m = Collections.synchronizedMap(new HashMap());
    ...
Set s = m.keySet();  // Needn't be in synchronized block
    ...
synchronized (m) {  // Synchronizing on m, not s!
    Iterator i = s.iterator(); // Must be in synchronized block
    while (i.hasNext())
        foo(i.next());
}
```

LinkedHashMap:

1. LinkedHashMap is the implementation of Map. It maintains insertion order.
2. LinkedHashMap allow null keys and values
3. It is not synchronized.
4. Map<K,V> map = new LinkedHashMap<K,V>();

TreeMap:

1.TreeMap is the implementation of Map and SortedMap. It maintains ascending order.

2.TreeMap doesn't allow any null key or value.

3.It is not synchronized.

TreeMap<K,V> treeMap= new TreeMap<K,V>();

Load Factor:

- The Load factor is a measure that decides when to **increase** the HashMap capacity to maintain the get() and put() operation complexity of **O(1)**. The default load factor of HashMap is **0.75f** (75% of the map size).

  load factor ratio = m/n

- Where,m is the number of entries in a hashmap.

  n is the total size of hashmap.
  The initial capacity of hashmap is=16
  The default load factor of hashmap=0.75
  According to the formula as mentioned above: 16*0.75=12

- It represents that 12^{th} key-value pair of hashmap will keep its size to 16. As soon as 13^{th} element (key-value pair) will come into the Hashmap, it will increase its size from default $\mathbf{2^4 = 16}$ buckets to $\mathbf{2^5 = 32}$ buckets.

Hashtable:

- It contains key, value pairs.
- Hashtable contains unique elements.
- Hashtable doesn't allow null key or value.
- Hashtable is synchronized.
- The initial default capacity of Hashtable class is 11 and loadFactor is 0.75.

  when the size of the Hashtable (number of elements) exceeds 3/4th of the capacity (8), the capacity of the Hashtable is doubled (22).

Hashtable() :It creates an empty hashtable having the initial default capacity and load factor.

Hashtable(int capacity):It accepts an integer parameter and creates a hash table that contains a specified initial capacity.

Hashtable(int capacity, float loadFactor):It is used to create a hash table having the specified initial capacity and loadFactor.

Hashtable(Map<? extends K,? extends V> t): It creates a new hash table with the same mappings as the given Map.

- **Set Interface:**
  It extends Collection interface and doesn't allow insertion of duplicate elements.
  It has two sub interfaces, SortedSet and NavigableSet.
- SortedSet interface extends Set interface and arranges added elements in an ascending order.
- NavigabeSet interface extends SortedSet interface, and allows retrieval of elements based on the closest match to a given value or values.
- The three main implementations of the Set interface are HashSet, TreeSet, and LinkedHashSet.

HashSet:

- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet is non synchronized.
- HashSet doesn't maintain the insertion order.
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.

HashSet():It is used to construct a default HashSet.

HashSet(int capacity):It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.

HashSet(int capacity, float loadFactor):It is used to initialize the capacity of the hash set to the given integer value capacity and the specified load factor.

HashSet(Collection<? extends E> c):It is used to initialize the hash set by using the elements of the collection c.

```
class HashSetTest{
 public static void main(String args[]){
   HashSet<String> set=new HashSet();
        set.add("Hyderabad");
        set.add("Bangalore");
        set.add("Hyderabad");
        set.add("Mumbai");
        set.add("Hyderabad");
        Iterator<String> i=set.iterator();
        while(i.hasNext())
        {
        System.out.println(i.next());
        }
   }
}
```

SN	Modifier & Type	Method	Description
1)	boolean	add(E e)	It is used to add the specified element to this set if it is not already present.
2)	void	clear()	It is used to remove all of the elements from the set.
3)	object	clone()	It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
4)	boolean	contains(Object o)	It is used to return true if this set contains the specified element.
5)	boolean	isEmpty()	It is used to return true if this set contains no elements.
6)	Iterator<E>	iterator()	It is used to return an iterator over the elements in this set.
7)	boolean	remove(Object o)	It is used to remove the specified element from this set if it is present.
8)	int	size()	It is used to return the number of elements in the set.
9)	Spliterator<E>	spliterator()	It is used to create a late-binding and fail-fast Spliterator over the elements in the set.

LinkedHashSet:

- LinkedHashSet is maintains insertion order.
- LinkedHashSet is non synchronized.
- It provides all set operations and permits null elements.

```
class LinkedHashSetTest{
 public static void main(String args[]){
      LinkedHashSet<String> set=new LinkedHashSet<>();
           set.add("Hyderabad");
           set.add("Bangalore");
           set.add("Hyderabad");
           set.add("Mumbai");
           set.add("Delhi");
           Iterator<String> i=set.iterator();
           while(i.hasNext())
           {
           System.out.println(i.next());
           }
    }
}
```

TreeSet:

- TreeSet doesn't allow null element.
- TreeSet is non synchronized.
- TreeSet maintains ascending order of values.
- TreeSet access and retrieval times are quiet fast.

```
class TreeSetTest{
 public static void main(String args[]){
  TreeSet<String> al=new TreeSet<String>();
  al.add("Ramu");
  al.add("Surya");
  al.add("Balu");
  al.add("Ajay");
  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```