JAVA Collections Framework

The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

A Collection represents a single unit of objects, i.e., a group.

Framework in Java:

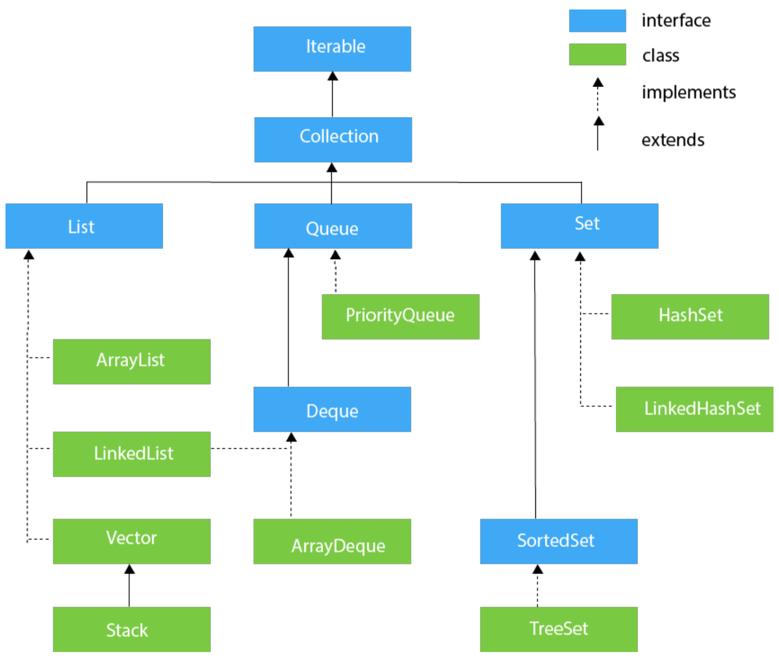
- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

Collection Framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

- Interfaces and its implementations, i.e., classes
- Algorithm

Hierarchy of Collection Framework



Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
6	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.

7	public int size()	It returns the total number of elements in the collection.
8	public void clear()	It removes the total number of elements from the collection.
9	public boolean contains(Object element)	It is used to search an element.
10	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
11	public Iterator iterator()	It returns an iterator.
12	public Object[] toArray()	It converts collection into array.
13	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
14	public boolean isEmpty()	It checks if collection is empty.
15	default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
16	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
17	default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.
18	public boolean equals(Object element)	It matches two collections.
19	public int hashCode()	It returns the hash code number of the collection.

Iterator interface

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

Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.

Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

```
Iterator<T> iterator()
```

It returns the iterator over the elements of type T.

Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

```
List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
List <data-type> list4 = new Stack();
```

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The classes that implement the List interface are given below.

ArrayList
The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```
import java.util.*;
class TestJavaCollection1{
        public static void main(String args[]){
                ArrayList<String> list=new ArrayList<String>();//Creating arraylist
                list.add("Rohit");//Adding object in arraylist
                list.add("Virat");
                list.add("Sachin");
                list.add("Sourav");
                //Traversing list through Iterator
                Iterator itr=list.iterator();
                while(itr.hasNext()){
                        System.out.println(itr.next());
                }
        }
}
```

LinkedList
LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.
Consider the following example.

```
import java.util.*;
public class TestJavaCollection2{
        public static void main(String args[]){
                LinkedList<String> al=new LinkedList<String>();
                al.add("Rohit");
                al.add("Virat");
                al.add("Sachin");
                al.add("Sourav");
                Iterator<String> itr=al.iterator();
                while(itr.hasNext()){
                        System.out.println(itr.next());
                }
        }
}
```

Vector
Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.
Consider the following example.

```
import java.util.*;
public class TestJavaCollection3{
        public static void main(String args[]){
                Vector<String> v=new Vector<String>();
                v.add("Ayush");
                v.add("Amit");
                v.add("Ashish");
                v.add("Garima");
                Iterator<String> itr=v.iterator();
                while(itr.hasNext()){
                        System.out.println(itr.next());
                }
        }
}
```

Stack
The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.
Consider the following example.

```
import java.util.*;
public class TestJavaCollection4{
        public static void main(String args[]){
                Stack<String> stack = new Stack<String>();
                stack.push("Ayush");
                stack.push("Garvit");
                stack.push("Amit");
                stack.push("Ashish");
                stack.push("Garima");
                stack.pop();
                Iterator<String> itr=stack.iterator();
                while(itr.hasNext()){
                        System.out.println(itr.next());
                }
        }
}
```

Queue Interface
Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

```
Queue<String> q1 = new PriorityQueue();
Queue<String> q2 = new ArrayDeque();
```

There are various classes that implement the Queue interface, some of them are given below.

PriorityQueue
The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection5{
        public static void main(String args[]){
                PriorityQueue<String> queue=new PriorityQueue<String>();
                queue.add("Amit Sharma");
                queue.add("Vijay Raj");
                queue.add("JaiShankar");
                queue.add("Raj");
                System.out.println("head:"+queue.element());
                System.out.println("head:"+queue.peek());
                System.out.println("iterating the queue elements:");
                Iterator itr=queue.iterator();
                while(itr.hasNext()){
                        System.out.println(itr.next());
                }
                queue.remove();
                queue.poll();
                System.out.println("after removing two elements:");
                Iterator<String> itr2=queue.iterator();
                while(itr2.hasNext()){
                        System.out.println(itr2.next());
                }
        }
}
```

Deque Interface
Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.
Deque can be instantiated as:

```
Deque d = new ArrayDeque();
```

ArrayDeque
ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection6{
        public static void main(String[] args) {
                //Creating Deque and adding elements
                Deque<String> deque = new ArrayDeque<String>();
                deque.add("Gautam");
                deque.add("Karan");
                deque.add("Ajay");
                //Traversing elements
                for (String str : deque) {
                        System.out.println(str);
                }
        }
}
```

Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

```
Set<data-type> s1 = new HashSet<data-type>();
Set<data-type> s2 = new LinkedHashSet<data-type>();
Set<data-type> s3 = new TreeSet<data-type>();
```

HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection7{
        public static void main(String args[]){
                //Creating HashSet and adding elements
                HashSet<String> set=new HashSet<String>();
                set.add("Ravi");
                set.add("Vijay");
                set.add("Ravi");
                set.add("Ajay");
                //Traversing elements
                Iterator<String> itr=set.iterator();
                while(itr.hasNext()){
                        System.out.println(itr.next());
                }
        }
}
```

LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection8{
        public static void main(String args[]){
                LinkedHashSet<String> set=new LinkedHashSet<String>();
                set.add("Ravi");
                set.add("Vijay");
                set.add("Ravi");
                set.add("Ajay");
                Iterator<String> itr=set.iterator();
                while(itr.hasNext()){
                        System.out.println(itr.next());
                }
        }
}
```

SortedSet Interface

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

```
SortedSet<data-type> set = new TreeSet();
```

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

```
import java.util.*;
```

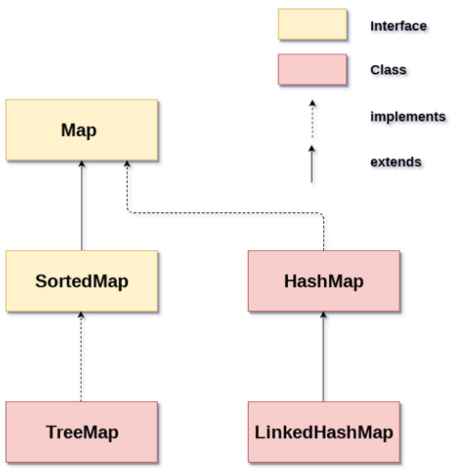
```
public class TestJavaCollection9{
        public static void main(String args[]){
                //Creating and adding elements
                TreeSet<String> set=new TreeSet<String>();
                set.add("Ravi");
                set.add("Vijay");
                set.add("Ravi");
                set.add("Ajay");
                //traversing elements
                Iterator<String> itr=set.iterator();
                while(itr.hasNext()){
                        System.out.println(itr.next());
                }
        }
}
```

Java 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.

Java Map Hierarchy
There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

The hierarchy of Java Map is given below:



A Map can't be traversed, so you need to convert it into Set using keySet() or entrySet() method.

Class	Description
HashMap	HashMap is the implementation of Map, but it doesn't maintain any order.
LinkedHashMap	LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order.
TreeMap	TreeMap is the implementation of Map and SortedMap. It maintains ascending order.

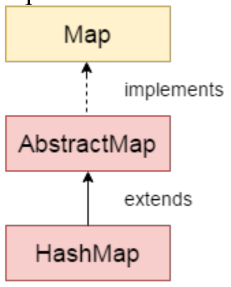
Useful methods of Map interface

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.
V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).

V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)	It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null.
V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.
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.
Collectionvalues()	It returns a collection view of the values contained in the map.
int size()	This method returns the number of entries in the map.

Java HashMap

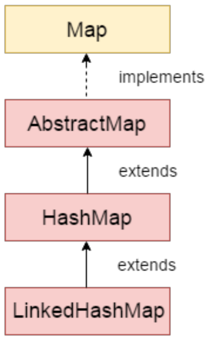
Java HashMap class implements the Map interface which allows us to store key and value pair, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.
HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as `HashMap<K,V>`, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.



```
import java.util.*;
public class HashMapExample1{
 public static void main(String args[]){
   HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap
   map.put(1,"Mango");  //Put elements in Map
   map.put(2,"Apple");
   map.put(3,"Banana");
   map.put(4,"Grapes");

   System.out.println("Iterating Hashmap...");
   for(Map.Entry m : map.entrySet()){
    System.out.println(m.getKey()+" "+m.getValue());
   }
 }
}
```

Java LinkedHashMap class

Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

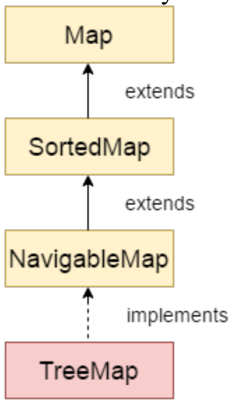


```
import java.util.*;
class LinkedHashMap1{
 public static void main(String args[]){

  LinkedHashMap<Integer,String> hm=new LinkedHashMap<Integer,String>();

  hm.put(100,"Amit");
  hm.put(101,"Vijay");
  hm.put(102,"Rahul");

for(Map.Entry m:hm.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```

Java TreeMap Class

Java TreeMap class is a red-black tree-based implementation. It provides an efficient means of storing key-value pairs in sorted order.

The java.util package contains the Java TreeMap class, which is a component of the Java Collections Framework. It extends the AbstractMap class and implements the NavigableMap interface. TreeMap is an effective red-black tree-based solution that sorts key-value pairs. TreeMap works well in situations where ordered key-value pairs are necessary since it preserves ascending order. It also only includes distinct components, guaranteeing that every key corresponds to a single value. TreeMap can have more than one null value, even if it cannot have a null key.



```
import java.util.*;
class TreeMap1{
 public static void main(String args[]){
   TreeMap<integer ,string> map=new TreeMap<integer ,string>();
      map.put(100,"Amit");
      map.put(102,"Ravi");
      map.put(101,"Vijay");
      map.put(103,"Rahul");

      for(Map.Entry m:map.entrySet()){
       System.out.println(m.getKey()+" "+m.getValue());
      }
 }
}
```