Collections in Java

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).

What is Collection in Java

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

What is a framework in Java

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

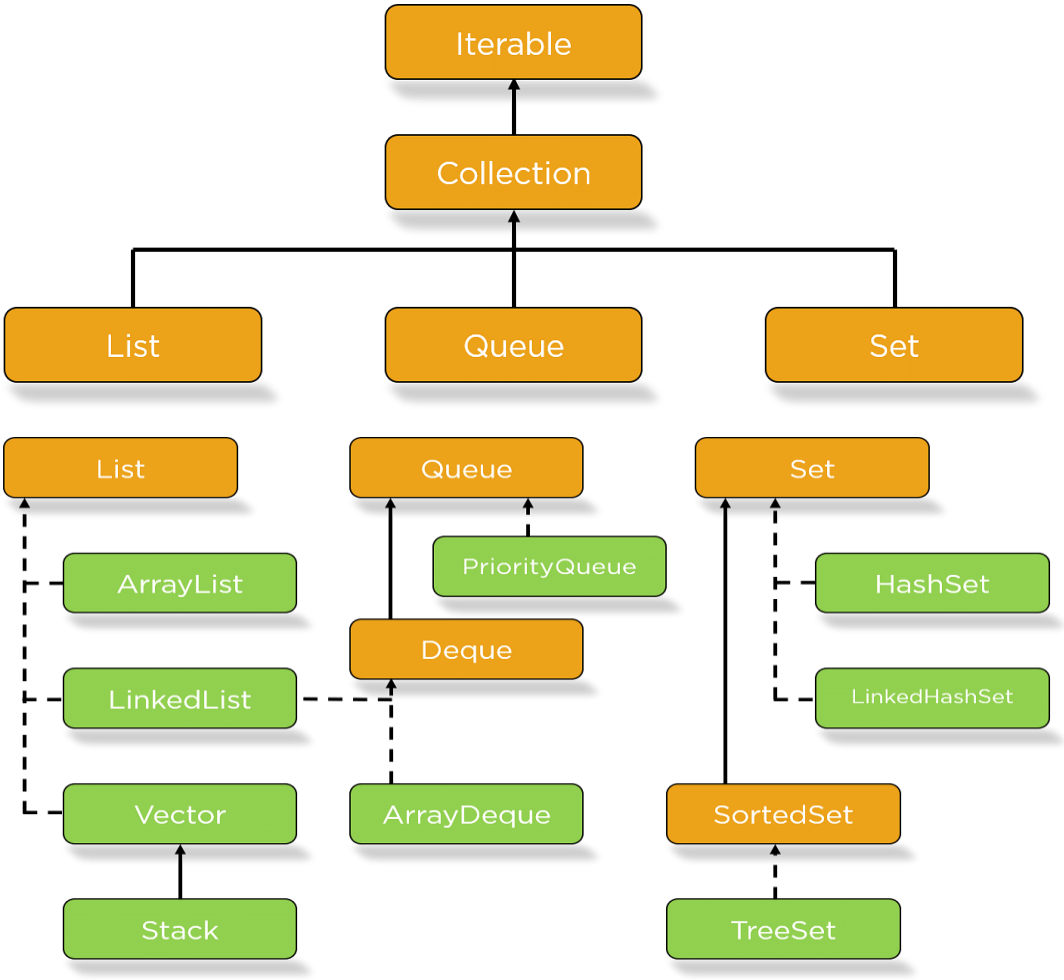
What is Collection framework

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

1. Interfaces and its implementations, i.e., classes
2. Algorithm

Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The **java.util** package contains all the classes and interfaces for the Collection framework.



java Collections Interface Methods The table below describes the methods available to use against Java Collections for data Manipulation Jobs.

Method	Description
add()	Add objects to collection
isEmpty()	Returns true if collection is empty
clear()	Removes all elements from the collection
remove()	Remove a selected object
size()	Find the number of elements
stream()	Return Sequential elements
max()	Return max value in the collection
retainAll()	Retains elements in the collection

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 :

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

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.

```
1. import java.util.*;
2. class TestJavaCollection1{
3. public static void main(String args[]){
4. ArrayList<String> list=new ArrayList<String>();//Creating arraylist
5. list.add("Ravi");//Adding object in array
6. list
7. list.add("Vijay");
8. list.add("Ravi");
9. list.add("Ajay");
10.//Traversing list through Iterator
11.Iterator itr=list.iterator();
12.while(itr.hasNext()){
13.System.out.println(itr.next());
14.}
15.}
16.}
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

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.

```
1. import java.util.*;
2. public class TestJavaCollection2{
3. public static void main(String args[]){
4. LinkedList<String> al=new LinkedList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. Iterator<String> itr=al.iterator();
10.while(itr.hasNext()){
11.System.out.println(itr.next());
12.}
13.}
14.}
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

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.

```
1. import java.util.*;
2. public class TestJavaCollection3{
3. public static void main(String args[]){
4. Vector<String> v=new Vector<String>();
```

```
5. v.add("Ayush");
6. v.add("Amit");
7. v.add("Ashish");
8. v.add("Garima");
9. Iterator<String> itr=v.iterator();
10.while(itr.hasNext()){
11.System.out.println(itr.next());
12.}
13.}
14.}
```

Output:

```
Ayush
Amit
Ashish
Garima
```

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:

```
1. Set<data-type> s1 = new HashSet<data-type>();
2. Set<data-type> s2 = new LinkedHashSet<data-type>();
3. 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.

```
1. import java.util.*;
2. public class TestJavaCollection7{
3. public static void main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=new HashSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
```

```
9.  set.add("Ajay");
10.//Traversing elements
11.Iterator<String> itr=set.iterator();
12.while(itr.hasNext()){
13.System.out.println(itr.next());
14.}
15.}
16.}
```

Output:

```
Vijay
Ravi
Ajay
```

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:

```
1.  import java.util.*;
2.  public class TestJavaCollection9{
3.  public static void main(String args[]){
4.  //Creating and adding elements
5.  TreeSet<String> set=new TreeSet<String>();
6.  set.add("Ravi");
7.  set.add("Vijay");
8.  set.add("Ravi");
9.  set.add("Ajay");
10.//traversing elements
11.Iterator<String> itr=set.iterator();
12.while(itr.hasNext()){
13.System.out.println(itr.next());
14.}
15.}
16.}
```

Output:

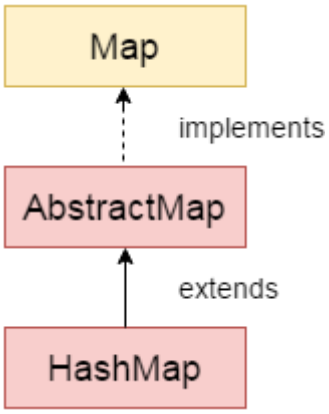
```
Ajay
Ravi
Vijay
```

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 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.

Points to remember

- Java HashMap contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap is non synchronized.
- Java HashMap maintains no order.

void clear()	It is used to remove all of the mappings from this map.
boolean isEmpty()	It is used to return true if this map contains no key-value mappings.
Object clone()	It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned.

Set entrySet()	It is used to return a collection view of the mappings contained in this map.
Set keySet()	It is used to return a set view of the keys contained in this map.
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 remove(Object key)	It is used to delete an entry for the specified key.
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.
V get(Object key)	This method returns the object that contains the value associated with the key.
boolean isEmpty()	This method returns true if the map is empty; returns false if it contains at least one key.
V replace(K key, V value)	It replaces the specified value for a specified key.
int size()	This method returns the number of entries in the map.

Java HashMap Example:

```
1. import java.util.*;
2. public class HashMapExample1{
3.  public static void main(String args[]){
4.    HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating Hash Map
5.    map.put(1,"Mango");  //Put elements in Map
6.    map.put(2,"Apple");
7.    map.put(3,"Banana");
8.    map.put(4,"Grapes");
9.
10.   System.out.println("Iterating Hashmap...");
11.   for(Map.Entry m : map.entrySet()){
12.    System.out.println(m.getKey()+" "+m.getValue());
13.   }
14. }
15. }
```

Test it Now

```
Iterating Hashmap...
1 Mango
```

```
2 Apple
3 Banana
4 Grapes
```

Java Hashtable class

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

Points to remember

- A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
- Java Hashtable class contains unique elements.
- Java Hashtable class doesn't allow null key or value.
- Java Hashtable class is synchronized.

ava Hashtable Example

```
1. import java.util.*;
2. class Hashtable1{
3.  public static void main(String args[]){
4.   Hashtable<Integer,String> hm=new Hashtable<Integer,String>();
5.
6.   hm.put(100,"Amit");
7.   hm.put(102,"Ravi");
8.   hm.put(101,"Vijay");
9.   hm.put(103,"Rahul");
10.
11.  for(Map.Entry m:hm.entrySet()){
12.   System.out.println(m.getKey()+" "+m.getValue());
13.  }
14. }
15. }
```

Test it Now

Output:

```
103 Rahul
102 Ravi
101 Vijay
100 Amit
```

Java Hashtable Example: remove()

```
1. import java.util.*;
2. public class Hashtable2 {
3.   public static void main(String args[]) {
4.  Hashtable<Integer,String> map=new Hashtable<Integer,String>();
5.     map.put(100,"Amit");
```

```
6.      map.put(102,"Ravi");
7.      map.put(101,"Vijay");
8.      map.put(103,"Rahul");
9.      System.out.println("Before remove: "+ map);
10.       // Remove value for key 102
11.      map.remove(102);
12.      System.out.println("After remove: "+ map);
13.   }
14. }
```

Output:

```
Before remove: {103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}
After remove: {103=Rahul, 101=Vijay, 100=Amit}
```

Accessing a Java Collection via a Iterator

The java collection framework often we want to cycle through the elements. For example, we might want to display each element of a collection. The java provides an interface **Iterator** that is available inside the **java.util** package to cycle through each element of a collection.

🔔 The **Iterator** allows us to move only forward direction.

🔔 The **Iterator** does not support the replacement and addition of new elements.

We use the following steps to access a collection of elements using the Iterator.

JavaIteratorExample.java

```
1.  import java.io.*;
2.  import java.util.*;
3.
4.  public class JavaIteratorExample {
5.     public static void main(String[] args)
6.     {
7.        ArrayList<String> cityNames = new ArrayList<String>();
8.
9.        cityNames.add("Delhi");
10.       cityNames.add("Mumbai");
11.       cityNames.add("Kolkata");
12.       cityNames.add("Chandigarh");
13.       cityNames.add("Noida");
```

```
14.
15.         // Iterator to iterate the cityNames
16.         Iterator iterator = cityNames.iterator();
17.
18.         System.out.println("CityNames elements : ");
19.
20.         while (iterator.hasNext())
21.             System.out.print(iterator.next() + " ");
22.
23.         System.out.println();
24.     }
25. }
```

Output:

```
CityNames elements:
Delhi Mumbai Kolkata Chandigarh Noida
```

Java Comparator interface

Java Comparator interface is used to order the objects of a user-defined class.

This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).

It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Methods of Java Comparator Interface

Method	Description
public int compare(Object obj1, Object obj2)	It compares the first object with the second object.
public boolean equals(Object obj)	It is used to compare the current object with the specified object.

Java Comparator Example (Non-generic Old Style)

Let's see the example of sorting the elements of List on the basis of age and name. In this example, we have created 4 java classes:

1. Student.java
2. AgeComparator.java
3. NameComparator.java

4. Simple.java

Student.java

```
class Student{
int rollno;
String name;
int age;
Student(int rollno,String name,int age){
this.rollno=rollno;
this.name=name;
this.age=age;
}
}
```

AgeComparator.java

```
import java.util.*;
class AgeComparator implements Comparator<Student>{
public int compare(Student s1,Student s2){
if(s1.age==s2.age)
return 0;
else if(s1.age>s2.age)
return 1;
else
return -1;
}
}
```

NameComparator.java

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

```
import java.util.*;
class NameComparator implements Comparator<Student>{
public int compare(Student s1,Student s2){
return s1.name.compareTo(s2.name);
}
}
```

Simple.java

In this class, we are printing the values of the object by sorting on the basis of name and age.

```
import java.util.*;
import java.io.*;
class Simple{
```

```
4.  public static void main(String args[]){
5.
6.  ArrayList<Student> al=new ArrayList<Student>();
7.  al.add(new Student(101,"Vijay",23));
8.  al.add(new Student(106,"Ajay",27));
9.  al.add(new Student(105,"Jai",21));
10.   System.out.println("Sorting by Name");
11.
12. Collections.sort(al,new NameComparator());
13. for(Student st: al){
14. System.out.println(st.rollno+" "+st.name+" "+st.age);
15. }
16.
17. System.out.println("Sorting by age");
18.
19. Collections.sort(al,new AgeComparator());
20. for(Student st: al){
21. System.out.println(st.rollno+" "+st.name+" "+st.age);
22. }
23. }
24. }
```

```
Sorting by Name
106 Ajay 27
105 Jai 21
101 Vijay 23

Sorting by age
105 Jai 21
101 Vijay 23
106 Ajay 27
```

Java Comparable interface

Java Comparable interface is used to order the objects of the user-defined class. This interface is found in java.lang package and 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. For example, it may be rollno, name, age or anything else.

compareTo(Object obj) method

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.

Java Comparable Example

Let's see the example of the Comparable interface that sorts the list elements on the basis of age.

File: Student.java

```
1. class Student implements Comparable<Student>{
2. int rollno;
3. String name;
4. int age;
5. Student(int rollno,String name,int age){
6. this.rollno=rollno;
7. this.name=name;
8. this.age=age;
9. }
10.  public int compareTo(Student st){
11. if(age==st.age)
12. return 0;
13. else if(age>st.age)
14. return 1;
15. else
16. return -1;
17. }
18. }
```

File: TestSort1.java

```
1. import java.util.*;
2. public class TestSort1{
3. public static void main(String args[]){
4. ArrayList<Student> al=new ArrayList<Student>();
5. al.add(new Student(101,"Vijay",23));
6. al.add(new Student(106,"Ajay",27));
7. al.add(new Student(105,"Jai",21));
8.
9. Collections.sort(al);
10. for(Student st:al){
11. System.out.println(st.rollno+" "+st.name+" "+st.age);
12. }
13. }
14. }
```

```
105 Jai 21
101 Vijay 23
106 Ajay 27
```

Difference between Comparable and Comparator

Comparable and Comparator both are interfaces and can be used to sort collection elements.

However, there are many differences between Comparable and Comparator interfaces that are given below.

Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.