

Unit-4

JAVA

COLLECTION

FRAMEWORK

# 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*

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

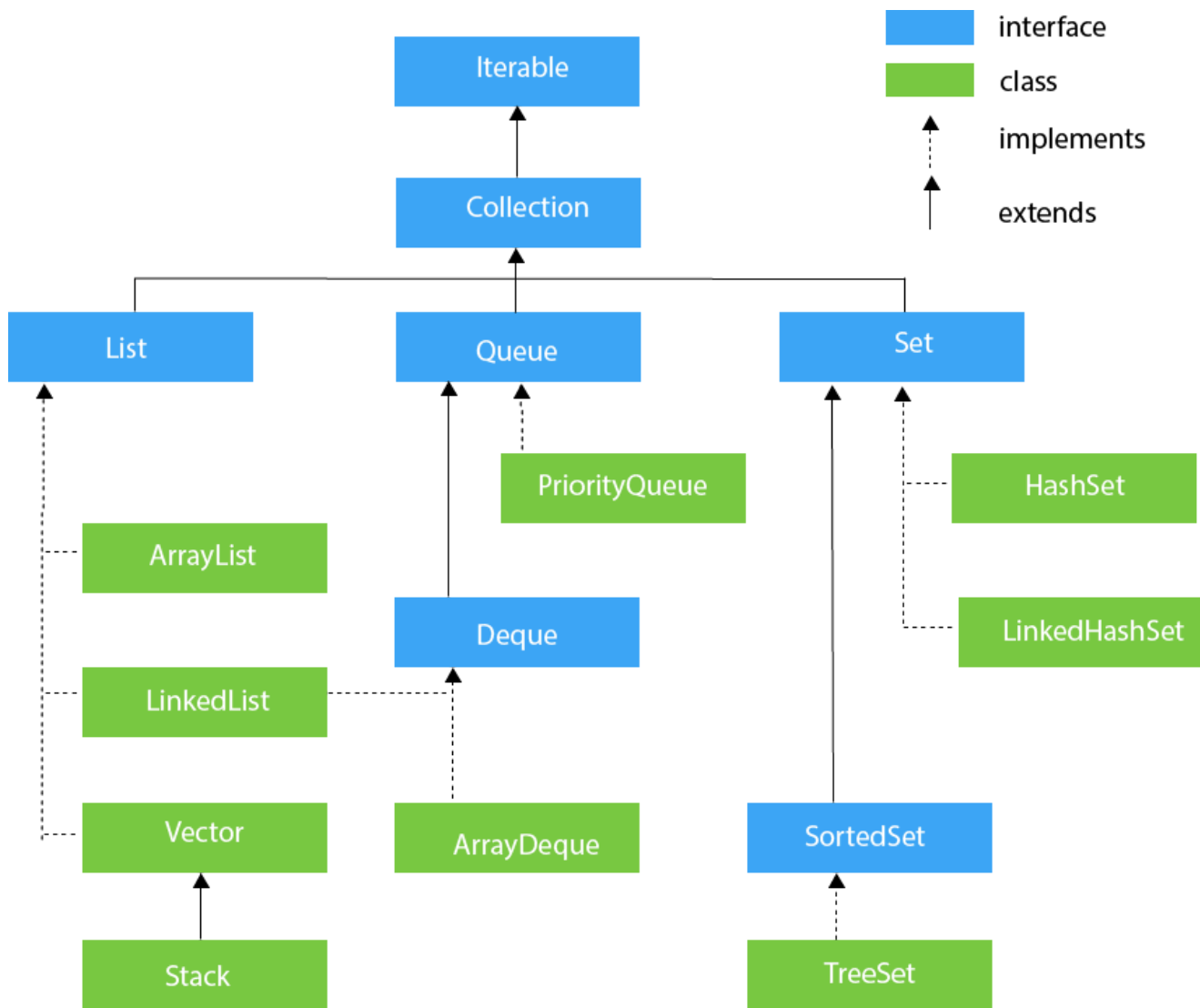
## *What is Collection framework*

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.



## 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.,

1. `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 :

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();

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.

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 arraylist
6. list.add("Vijay");
7. list.add("Ravi");
8. list.add("Ajay");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. **while**(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. }
15. }

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

```

## 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.

```

1. import java.util.*;

```

```

2. public class TestJavaCollection4{
3. public static void main(String args[]){
4. Stack<String> stack = new Stack<String>();
5. stack.push("Ayush");
6. stack.push("Garvit");
7. stack.push("Amit");
8. stack.push("Ashish");
9. stack.push("Garima");
10. stack.pop();
11. Iterator<String> itr=stack.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

```

Output:

```

Ayush
Garvit
Amit
Ashish

```

## 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:

```

1. Queue<String> q1 = new PriorityQueue();
2. 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.

```

1. import java.util.*;
2. public class TestJavaCollection5{
3. public static void main(String args[]){
4. PriorityQueue<String> queue=new PriorityQueue<String>();
5. queue.add("Amit Sharma");
6. queue.add("Vijay Raj");
7. queue.add("JaiShankar");
8. queue.add("Raj");
9. System.out.println("head:"+queue.element());
10. System.out.println("head:"+queue.peek());
11. System.out.println("iterating the queue elements:");
12. Iterator itr=queue.iterator();

```

```

13. while(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. queue.remove();
17. queue.poll();
18. System.out.println("after removing two elements:");
19. Iterator<String> itr2=queue.iterator();
20. while(itr2.hasNext()){
21. System.out.println(itr2.next());
22. }
23. }
24. }

```

Output:

ADVERTISEMENT

```

head:Amit Sharma
head:Amit Sharma
iterating the queue elements:
Amit Sharma
Raj
JaiShankar
Vijay Raj
after removing two elements:
Raj
Vijay Raj

```

## 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:

```

1. 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.

```

1. import java.util.*;
2. public class TestJavaCollection6{
3. public static void main(String[] args) {
4. //Creating Deque and adding elements
5. Deque<String> deque = new ArrayDeque<String>();
6. deque.add("Cautam");
7. deque.add("Karan");
8. deque.add("Ajay");
9. //Traversing elements
10. for (String str : deque) {
11. System.out.println(str);

```

```
12. }  
13. }  
14. }
```

Output:

```
Gautam  
Karan  
Ajay
```

## 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
```

## 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.

```
1. import java.util.*;
2. public class TestJavaCollection8{
3.     public static void main(String args[]){
4.         HashSet<String> set=new HashSet<String>();
5.         set.add("Ravi");
6.         set.add("Vijay");
7.         set.add("Ravi");
8.         set.add("Ajay");
9.         Iterator<String> itr=set.iterator();
10.        while(itr.hasNext()){
11.            System.out.println(itr.next());
12.        }
13.    }
14. }
```

Output:

```
Ravi
Vijay
Ajay
```

## 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:

```
1. 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:

```
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
```

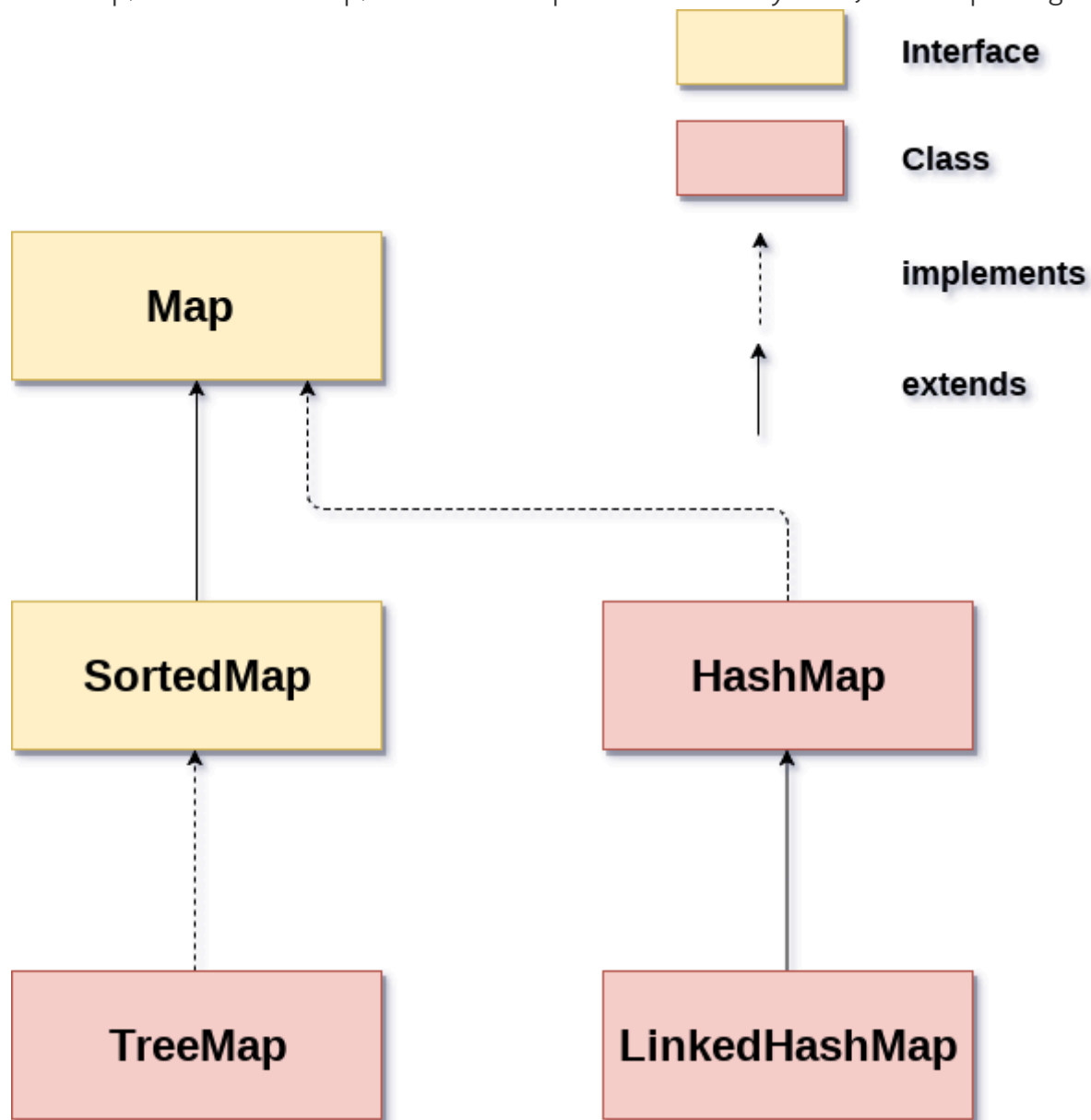
## Java Map Interface

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.

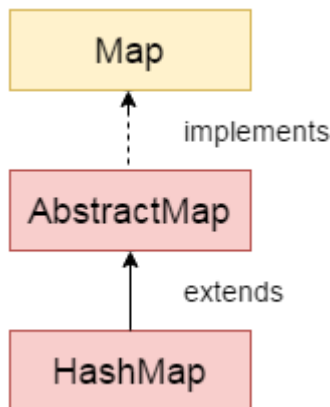
## Java Map Hierarchy

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below:



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.

## 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

- o Java HashMap contains values based on the key.
- o Java HashMap contains only unique keys.
- o Java HashMap may have one null key and multiple null values.
- o Java HashMap is non synchronized.
- o Java HashMap maintains no order.
- o The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

### Hierarchy of HashMap class

As shown in the above figure, HashMap class extends AbstractMap class and implements Map interface.