

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

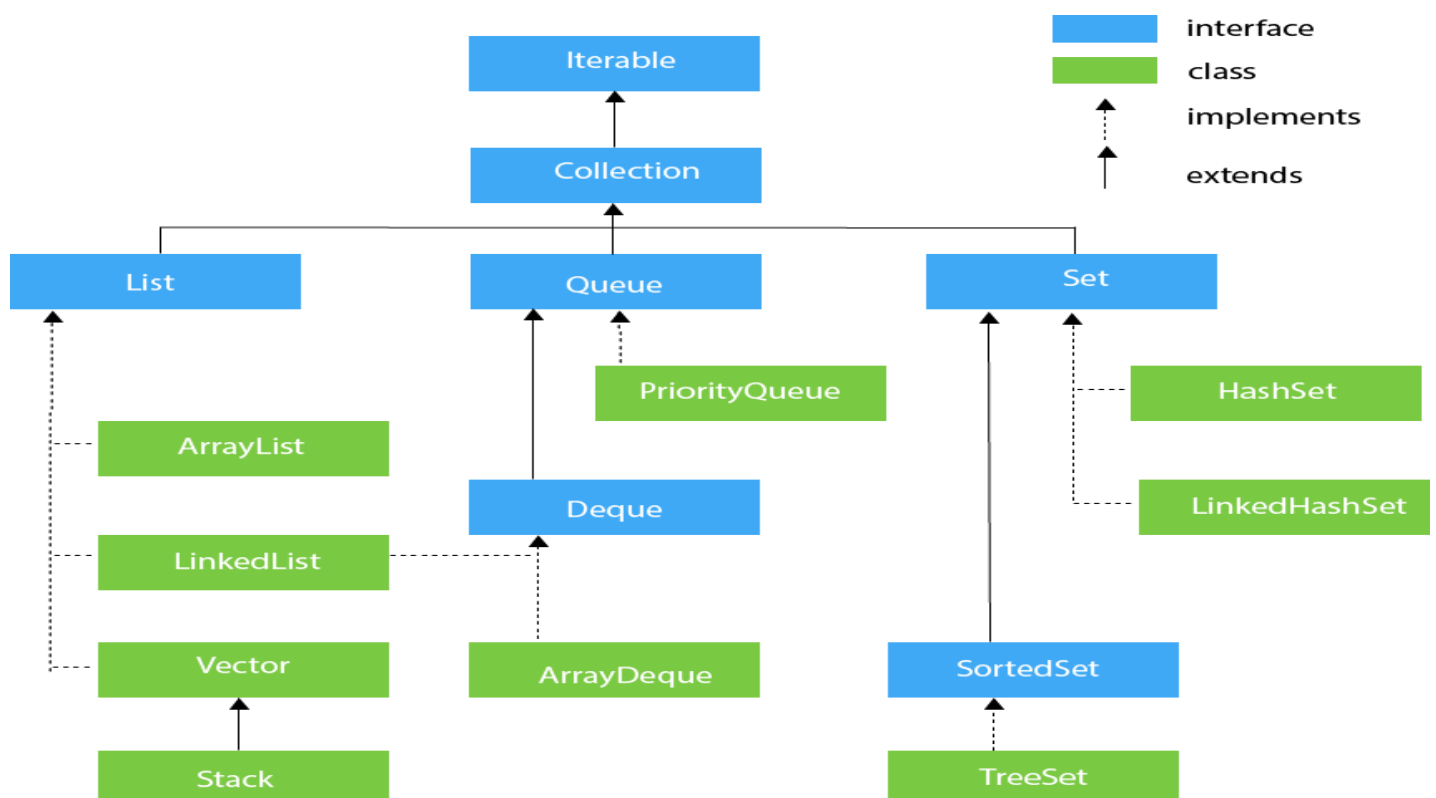
It provides readymade architecture.

It represents a set of classes and interfaces.

It is optional.

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.



Collection interface

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("Ravi");//Adding object in arraylist
```

```
list.add("Vijay");
```

```
list.add("Ravi");
```

```
list.add("Ajay");
```

```
//Traversing list through Iterator
```

```
Iterator itr=list.iterator();
```

```
while(itr.hasNext()){
```

```
System.out.println(itr.next());
```

```
}
```

```
}
```

```
}
```

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.

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

```
public class TestJavaCollection2{
```

```
public static void main(String args[]){

LinkedList<String> al=new LinkedList<String>();

al.add("Ravi");

al.add("Vijay");

al.add("Ravi");

al.add("Ajay");

Iterator<String> itr=al.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

}

}

}
```

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.

```
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());
```

```
}
```

```
}
```

```
}
```

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.

```
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());

}

}

}
```

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:

```
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());
```

```
}
```

```
}
```

```
}
```

Output:

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:

```
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);

        }

    }

}
```

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:

```
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());
```

```
}
```

```
}
```

```
}
```

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.

```
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());

        }
```

```
}
```

```
}
```

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:

```
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());
```

```
}
```

```
}
```

```
}
```

Output:

Ajay

Ravi

Vijay