**-: Collection :-**

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.

The collection framework was designed to meet several goals, such as −

* The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) were to be highly efficient.
* The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.

**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

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following −

* **Interfaces** − These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
* **Implementations, i.e., Classes** − These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
* **Algorithms** − These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

**for any Data, Adding New Data, Modifying Existing Data, Removing Existing Data, Retrieving data, searching data**

**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.



## The Collections Interfaces

The collections framework defines several interfaces. This section provides an overview of each interface −

|  |  |
| --- | --- |
| **Sr.No.** | **Interface & Description** |
| 1 | The Collection Interface  This enables you to work with groups of objects; it is at the top of the collections hierarchy. |
| 2 | The List Interface  This extends **Collection** and an instance of List stores an ordered collection of elements. |
| 3 | The Set  This extends Collection to handle sets, which must contain unique elements. |
| 4 | The SortedSet  This extends Set to handle sorted sets. |
| 5 | The Map  This maps unique keys to values. |
| 6 | The Map.Entry  This describes an element (a key/value pair) in a map. This is an inner class of Map. |
| 7 | The Sorted Map  This extends Map so that the keys are maintained in an ascending order. |
| 8 | The Enumeration  This is legacy interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator. |

**Collection interface**

The Collection interface is the foundation upon which the collections framework is built. It declares the core methods that all collections will have. These methods are summarized in the following table.

Because all collections implement Collection, familiarity with its methods is necessary for a clear understanding of the framework. Several of these methods can throw an **UnsupportedOperationException**.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **boolean add(Object obj)**  Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates. |
| 2 | **void clear( )**  Removes all elements from the invoking collection. |
| 3 | **boolean contains(Object obj)**  Returns true if obj is an element of the invoking collection. Otherwise, returns false. |
| 4 | **boolean equals(Object obj)**  Returns true if the invoking collection and obj are equal. Otherwise, returns false. |
| 5 | **Iterator iterator( )**  Returns an iterator for the invoking collection. |
| 6 | **boolean remove(Object obj)**  Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false. |
| 7 | **boolean removeAll(Collection c)**  Removes all elements of c from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false. |

**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.,

1. Iterator<T> iterator()

It returns the iterator over the elements of type T.

**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

### Difference Between ArrayList and LinkedList in Java

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| This class uses a dynamic array to store the elements in it. With the introduction of [generics](https://www.geeksforgeeks.org/generics-in-java/), this class supports the storage of all types of objects. | This class uses a [doubly linked list](https://www.geeksforgeeks.org/doubly-linked-list/) to store the elements in it. Similar to the ArrayList, this class also supports the storage of all types of objects. |
| Manipulating ArrayList takes more time due to the internal implementation. Whenever we remove an element, internally, the array is traversed and the memory bits are shifted. | Manipulating LinkedList takes less time compared to ArrayList because, in a doubly-linked list, there is no concept of shifting the memory bits. The list is traversed and the reference link is changed. |
| This class implements a [List interface](https://www.geeksforgeeks.org/list-interface-java-examples/). Therefore, this acts as a list. | This class implements both the [List interface](https://www.geeksforgeeks.org/list-interface-java-examples/) and the [Deque interface](https://www.geeksforgeeks.org/deque-interface-java-example/). Therefore, it can act as a list and a deque. |
| This class works better when the application demands storing the data and accessing it. | This class works better when the application demands manipulation of the stored data and removing data. |

## 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.

Vector implements List Interface. Like ArrayList it also maintains insertion order but it is rarely used in non-thread environment as it is synchronized and due to which it gives poor performance in searching, adding, delete and update of its elements.

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 pop(), 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

**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:

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:

* 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

[ArrayList in java with example programs - Collections Framework (beginnersbook.com)](https://beginnersbook.com/2013/12/java-arraylist/)

[Java Comparator - javatpoint](https://www.javatpoint.com/Comparator-interface-in-collection-framework)

System.out.println("Example with Custom comparator with for Sorting");

List<Employee> empList = new ArrayList<>();

Employee e1 = new Employee(100,"Jhon", 30);

Employee e2 = new Employee(101,"Amenda", 35);

Employee e3 = new Employee(102,"Alexa", 40);

empList.add(e1);

empList.add(e2);

empList.add(e3);

Employee e4 = new Employee(103,"Jhon Abraham", 33);

empList.add(e4);

Comparator<Employee> nameComparator = Comparator.comparing( emp -> emp.getName());

System.out.println("before sorting by name : "+empList);

Collections.sort(empList, nameComparator);

System.out.println("Sorting by name comparing objects : "+empList);