**Collections Interface**

**Collection**

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

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

**Defintion:**

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**.

## **Example**

Following is an example to explain few methods from various class implementations of the above collection methods −

import java.util.\*;

public class CollectionsDemo {

public static void main(String[] args) {

// ArrayList

List a1 = new ArrayList();

a1.add("Zara");

a1.add("Mahnaz");

a1.add("Ayan");

System.out.println(" ArrayList Elements");

System.out.print("\t" + a1);

// LinkedList

List l1 = new LinkedList();

l1.add("Zara");

l1.add("Mahnaz");

l1.add("Ayan");

System.out.println();

System.out.println(" LinkedList Elements");

System.out.print("\t" + l1);

// HashSet

Set s1 = new HashSet();

s1.add("Zara");

s1.add("Mahnaz");

s1.add("Ayan");

System.out.println();

System.out.println(" Set Elements");

System.out.print("\t" + s1);

// HashMap

Map m1 = new HashMap();

m1.put("Zara", "8");

m1.put("Mahnaz", "31");

m1.put("Ayan", "12");

m1.put("Daisy", "14");

System.out.println();

System.out.println(" Map Elements");

System.out.print("\t" + m1);

}

}

## **Output**

ArrayList Elements

[Zara, Mahnaz, Ayan]

LinkedList Elements

[Zara, Mahnaz, Ayan]

Set Elements

[Ayan, Zara, Mahnaz]

Map Elements

{Daisy = 14, Ayan = 12, Zara = 8, Mahnaz = 31}

### **Methods of Collection interface**

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

|  |  |
| --- | --- |
| **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 | **boolean addAll(Collection c)**  Adds all the elements of c to the invoking collection. Returns true if the operation succeeds (i.e., the elements were added). Otherwise, returns false. |
| 3 | **void clear( )**  Removes all elements from the invoking collection. |
| 4 | **boolean contains(Object obj)**  Returns true if obj is an element of the invoking collection. Otherwise, returns false. |
| 5 | **boolean containsAll(Collection c)**  Returns true if the invoking collection contains all elements of **c**. Otherwise, returns false. |
| 6 | **boolean equals(Object obj)**  Returns true if the invoking collection and obj are equal. Otherwise, returns false. |
| 7 | **int hashCode( )**  Returns the hash code for the invoking collection. |
| 8 | **boolean isEmpty( )**  Returns true if the invoking collection is empty. Otherwise, returns false. |
| 9 | **Iterator iterator( )**  Returns an iterator for the invoking collection. |
| 10 | **boolean remove(Object obj)**  Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false. |
| 11 | **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. |
| 12 | **boolean retainAll(Collection c)**  Removes all elements from the invoking collection except those in c. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false. |
| 13 | **int size( )**  Returns the number of elements held in the invoking collection. |
| 14 | **Object[ ] toArray( )**  Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements. |
| 15 | **Object[ ] toArray(Object array[ ])**  Returns an array containing only those collection elements whose type matches that of array. |

## **Comparable Interface**

**Comparable in Java** is an object to compare itself with another object. It helps to sort the list of custom objects. The java.lang.Comparable interface should be implemented by a class in order to compare its instances. Array of objects implementing a comparable interface is sorted automatically by Arrays.sort and Collections.sort methods.

Comparable interface is present in java.lang package. The logic of sorting must be in the same class whose object you are going to sort. The class whose objects you want to sort must implement comparable interface. It provides single sorting sequences. This method can sort the data according to the natural sorting order. It affects the original class. i.e., actual class is altered. Implemented frequently in the API by: Calendar, Wrapper classes, Date, and  
String. All wrapper classes and String class implement comparable interface.

## **Method used in Comparable:**

### **CompareTo():**

[CompareTo() method](https://www.guru99.com/string-compareto-method-java.html) is used to perform natural sorting on string. The meaning of natural sorting is the sort order which applies on the object, e.g., numeric order for sorting integers, alphabetical order for String, etc.

The syntax of CompareTo() method is as follows:

int compareTo(T obj)

CompareTo() method compares the object with T obj.

**Output:**

* It returns 0 if the values are equal.
* In case, if the object has a lesser value, then this method returns a negative value.
* If the object has a higher value, it returns a positive value.

#### **CompareTo() Method Example:**

The following program of Java comparable example shows the comparison of two characters, “a” and “b”. Character “a” comes before “b” alphabetically.

Therefore, output is -1. Character “b” comes after “a” alphabetically. Hence output is 1. Character “a” and “b” both are equivalent. Hence output is 0.

public class Sample\_String {

public static void main(String[] args) {

String str\_Sample = "a";

System.out.println("Compare To 'a' b is : " + str\_Sample.compareTo("b"));

str\_Sample = "b";

System.out.println("Compare To 'b' a is : " + str\_Sample.compareTo("a"));

str\_Sample = "b";

System.out.println("Compare To 'b' b is : " + str\_Sample.compareTo("b"));

}

}

**Output**

Compare To 'a' b is : -1

Compare To 'b' a is : 1

Compare To 'b' b is : 0

**Comparator interface**

**Comparator in Java** is an object for comparing different objects of different classes. Comparator interface in Java is also used to arrange the objects of user-defined classes. It includes two important comparator interface methods known as compare (Object obj1, Object obj2) and equals (Object element).

Comparator interface is present in java.util package. The logic of sorting should be in separate class to write different sorting based on different attributes of objects. Class, whose objects you want to sort, do not need to implement a comparator interface. It provides multiple sorting sequences. This method sorts the data according to the customized sorting order. It is implemented to sort instances of third-party classes. The only implemented classes of Comparator are Collator and RuleBasedColator.

## **Method used in Comparator:**

### **Compare():**

Compare() enables you to order objects. To do this, you have to create a class that implements comparator interface. After this, you need to override it’s compare method.

The syntax of compare() method is as follows:

compare(Object obj1, Object obj2)

In the above syntax, obj1 and obj2 are two objects that you have to compare using compare() method.

**Output:**

* It returns a negative integer if the first argument is less than the second one.
* Returns zero if the first argument and second argument is equal.
* This method can return a positive integer, in case the first argument is greater than the second.

You have to ensure that the relation is transitive. For example, ((compare(a, b)>0) && (compare(b, c)>0)) which implies compare(a, c)>0.

#### **Compare Method Example:**

In the below program of Java comparator example, there are 6 variables. “x”, “y”, “p”, “q”, “r”, and “s”. The output is -1 as the value of “x”, 20 is less than the value of “y”, 30. The output is 0 because the value of “p”, 40 is equal to the value of “q”, 40.

import java.lang.Integer;

public class example {

public static void main(String args[])

{

int x = 20;

int y = 30;

// as 10 less than 20,

// Output will be a value less than zero

System.out.println(Integer.compare(x, y));

int p = 40;

int q = 40;

// as 30 equals 30,

// Output will be zero

System.out.println(Integer.compare(p, q));

int r = 20;

int s = 7;

// as 15 is greater than 8,

// Output will be a value greater than zero

System.out.println(Integer.compare(r, s));

}

}

**Output:**

-1

0

1