Collections Class in Java

Difficulty Level : Easy Last Updated : 25 Aug, 2021

Collections class is a member of the Java Collections Framework. The java.util.Collections package is the package that contains the Collections class. Collections class is basically used with the static methods that operate on the collections or return the collection. All the methods of this class throw the **NullPointerException** if the collection or object passed to the methods is null.

Syntax: Declaration

```
public class Collections
extends Object
```

Remember: *Object is the parent class of all the classes.*

Collections class fields

The collection class basically contains 3 fields as listed below which can be used to return immutable entities.

- EMPTY_LIST to get an immutable empty List
- EMPTY_SET to get an immutable empty Map
- EMPTY_MAP to get an immutable empty Set

Now let us do discuss methods that are present inside this class so that we can implement these inbuilt functionalities later on in our program. Below are the methods been listed below in a tabular format as shown below as follows:

Methods	Description
addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
asLifoQueue(Deque<T> deque)	This method returns a view of a Deque as a Last-in-first-out (Lifo) Queue.

binarySearch(List<? extends Comparable> list, T key)	This method searches the key using binary search in the specified list.
binarySearch(List<? extends T> list, T key, Comparator<? super T> c)	This method searches the specified list for the specified object using the binary search algorithm.
checkedCollection(Collection<E> c, Class<E> type)	This method returns a dynamically typesafe view of the specified collection.
checkedList(List<E> list, Class<E> type)	This method returns a dynamically typesafe view of the specified list.
checkedMap(Map<K,V> m, Class<K> keyType, Class<V> valueType)	This method returns a dynamically typesafe view of the specified map.
checkedNavigableMap (NavigableMap<K,V> m, Class<K> keyType, Class<V> valueType)	This method returns a dynamically typesafe view of the specified navigable map.
checkedNavigableSet (NavigableSet<E> s, Class<E> type)	This method returns a dynamically typesafe view of the specified navigable set.
checkedQueue(Queue<E> queue, Class<E> type)	This method returns a dynamically typesafe view of the specified queue.
checkedSet(Set<E> s, Class<E> type)	This method returns a dynamically typesafe view of the specified set.
checkedSortedMap(SortedMap<K, V> m, Class<K> keyType, Class<V> valueType)	This method returns a dynamically typesafe view of the specified sorted map.
checkedSortedSet(SortedSet<E> s, Class<E> type)	This method returns a dynamically typesafe view of the specified sorted set.
copy(List<? super T> dest, List<? extends T> src)	This method copies all of the elements from one list into another.
disjoint(Collection<?> c1, Collection<?> c2)	This method returns true if the two specified collections have no elements in common.
emptyEnumeration()	This method returns an enumeration that has no elements.

emptyIterator()	This method returns an iterator that has no elements.
emptyList()	This method returns an empty list (immutable).
emptyListIterator()	This method returns a list iterator that has no elements.
emptyMap()	This method returns an empty map (immutable).
emptyNavigableMap()	This method returns an empty navigable map (immutable).
emptyNavigableSet()	This method returns an empty navigable set (immutable).
emptySet()	This method returns an empty set (immutable).
emptySortedMap()	This method returns an empty sorted map (immutable).
emptySortedSet()	This method returns an empty sorted set (immutable).
enumeration(Collection<T> c)	This method returns an enumeration over the specified collection.
fill(List<? super T> list, T obj)	This method replaces all of the elements of the specified list with the specified element.
frequency(Collection<?> c, Object o)	This method returns the number of elements in the specified collection equal to the specified object.
indexOfSubList(List<?> source, List<?> target)	This method returns the starting position of the first occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
lastIndexOfSubList(List<?> source, List<?> target)	This method returns the starting position of the last occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
list(Enumeration<T> e)	This method returns an array list containing the elements returned by the specified enumeration in the order they are returned by the enumeration.
max(Collection<? extends T> coll)	This method returns the maximum element of the given collection, according to the natural ordering of its elements.
max(Collection<? extends T> coll, Comparator<? super T> comp)	This method returns the maximum element of the given collection, according to the order induced by the specified comparator.

min(Collection<? extends T> coll)	This method returns the minimum element of the given collection, according to the natural ordering of its elements.
min(Collection<? extends T> coll, Comparator<? super T> comp)	This method returns the minimum element of the given collection, according to the order induced by the specified comparator.
nCopies(int n, T o)	This method returns an immutable list consisting of n copies of the specified object.
newSetFromMap(Map<E, Boolean> map)	This method returns a set backed by the specified map.
replaceAll(List<T> list, T oldVal, T newVal)	This method replaces all occurrences of one specified value in a list with another.
reverse(List<?> list)	This method reverses the order of the elements in the specified list
reverseOrder()	This method returns a comparator that imposes the reverse of the natural ordering on a collection of objects that implement the Comparable interface.
reverseOrder(Comparator<T> cmp)	This method returns a comparator that imposes the reverse ordering of the specified comparator.
rotate(List<?> list, int distance)	This method rotates the elements in the specified list by the specified distance.
shuffle(List<?> list)	This method randomly permutes the specified list using a default source of randomness.
shuffle(List<?> list, Random rnd)	This method randomly permute the specified list using the specified source of randomness.
singletonMap(K key, V value)	This method returns an immutable map, mapping only the specified key to the specified value.
singleton(T o)	This method returns an immutable set containing only the specified object.
singletonList(T o)	This method returns an immutable list containing only the specified object.
sort(List<T> list)	This method sorts the specified list into ascending order,

	according to the natural ordering of its elements.
sort(List<T> list, Comparator<? super T> c)	This method sorts the specified list according to the order induced by the specified comparator.
swap(List<?> list, int i, int j)	This method swaps the elements at the specified positions in the specified list.
synchronizedCollection (Collection<T> c)	This method returns a synchronized (thread-safe) collection backed by the specified collection.
synchronizedList(List<T> list)	This method returns a synchronized (thread-safe) list backed by the specified list.
synchronizedMap(Map<K,V> m)	This method returns a synchronized (thread-safe) map backed by the specified map.
synchronizedNavigableMap (NavigableMap<K,V> m)	This method returns a synchronized (thread-safe) navigable map backed by the specified navigable map.
synchronizedNavigableSet (NavigableSet<T> s)	This method returns a synchronized (thread-safe) navigable set backed by the specified navigable set.
synchronizedSet(Set<T> s)	This method returns a synchronized (thread-safe) set backed by the specified set.
synchronizedSortedMap (SortedMap<K,V> m)	This method returns a synchronized (thread-safe) sorted map backed by the specified sorted map.
synchronizedSortedSet (SortedSet<T> s)	This method returns a synchronized (thread-safe) sorted set backed by the specified sorted set.
unmodifiableCollection (Collection<? extends T> c)	This method returns an unmodifiable view of the specified collection.
unmodifiableList(List<? extends T> list)	This method returns an unmodifiable view of the specified list.
unmodifiableNavigableMap (NavigableMap<K,? extends V> m)	This method returns an unmodifiable view of the specified navigable map.
unmodifiableNavigableSet (NavigableSet<T> s)	This method returns an unmodifiable view of the specified navigable set.
unmodifiableSet(Set<? extends T>	This method returns an unmodifiable view of the specified

s)	set.
unmodifiableSortedMap (SortedMap<K,? extends V> m)	This method returns an unmodifiable view of the specified sorted map.
unmodifiableSortedSet (SortedSet<T> s)	This method returns an unmodifiable view of the specified sorted set.

Now, we are done with listing all the methods so by ar we have a faint hint with us in perceiving how important are these methods when thinking about a global programming perspective. The important and frequently widely used methods while writing optimized code as you will see these methods somehow in nearly every java optimized code because of havoc usage of Collections class in java. So here more likely in any class we will not just be implementing the method but will also be discussing operations that can be performed so that one can have conceptual clarity and strong command while implementing the same. The operations that we will be discussing are as follows:

- Adding elements to the Collections
- Sorting a Collection
- Searching in a Collection
- Copying Elements
- Disjoint Collection

Operation 1: Adding elements to the Collections class object

The addAll() method of **java.util.Collections** class is used to add all the specified elements to the specified collection. Elements to be added may be specified individually or as an array.

Example

```
// Java Program to Demonstrate Adding Elements
// Using addAll() method

// Importing required classes
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

// Main class
```

```
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Creating a list
        // Declaring object of string type
        List<String> items = new ArrayList<>();

        // Adding elements (items) to the list
        items.add("Shoes");
        items.add("Toys");

        // Add one or more elements
        Collections.addAll(items, "Fruits", "Bat", "Ball");

        // Printing the list contents
        for (int i = 0; i < items.size(); i++) {
            System.out.print(items.get(i) + " ");
        }
    }
}
```

Output

```
Shoes Toys Fruits Bat Ball
```

Operation 2: Sorting a Collection

java.util.Collections.sort() is used to sort the elements present in the specified list of Collection in ascending order. java.util.Collections.reverseOrder() is used to sort in the descending order.

Example

```
// Java program to demonstrate sorting
// a Collections using sort() method

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

// Main Class
// SortingCollectionExample
class GFG {
```

```
    // Main driver method
    public static void main(String[] args)
    {
        // Creating a list
        // Declaring object of string type
        List<String> items = new ArrayList<>();

        // Adding elements to the list
        // using add() method
        items.add("Shoes");
        items.add("Toys");

        // Adding one or more elements using addAll()
        Collections.addAll(items, "Fruits", "Bat", "Mouse");

        // Sorting according to default ordering
        // using sort() method
        Collections.sort(items);

        // Printing the elements
        for (int i = 0; i < items.size(); i++) {
            System.out.print(items.get(i) + " ");
        }

        System.out.println();

        // Sorting according to reverse ordering
        Collections.sort(items, Collections.reverseOrder());

        // Printing the reverse order
        for (int i = 0; i < items.size(); i++) {
            System.out.print(items.get(i) + " ");
        }
    }
}
```

Output

```
Bat Fruits Mouse Shoes Toys
Toys Shoes Mouse Fruits Bat
```

Operation 3: Searching in a Collection

java.util.Collections.binarySearch() method returns the position of an object in a sorted list. To use this method, the list should be sorted in ascending order, otherwise, the result returned from the method will be wrong. If the element exists in the list, the method will return the position of the

element in the sorted list, otherwise, the result returned by the method would be the – **(insertion point where the element should have been present if exist)-1)**.

Example

```
// Java Program to Demonstrate Binary Search
// Using Collections.binarySearch()

// Importing required classes
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

// Main class
// BinarySearchOnACollection
public class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Creating a List
        // Declaring object of string type
        List<String> items = new ArrayList<>();

        // Adding elements to object
        // using add() method
        items.add("Shoes");
        items.add("Toys");
        items.add("Horse");
        items.add("Ball");
        items.add("Grapes");

        // Sort the List
        Collections.sort(items);

        // BinarySearch on the List
        System.out.println(
            "The index of Horse is "
            + Collections.binarySearch(items, "Horse"));

        // BinarySearch on the List
        System.out.println(
            "The index of Dog is "
            + Collections.binarySearch(items, "Dog"));
    }
}
```

Output

```
The index of Horse is 2
The index of Dog is -2
```

Operation 4: Copying Elements

The copy() method of **java.util.Collections** class is used to copy all the elements from one list into another. After the operation, the index of each copied element in the destination list will be identical to its index in the source list. The destination list must be at least as long as the source list. If it is longer, the remaining elements in the destination list are unaffected.

Example

```
// Java Program to Demonstrate Copying Elements
// Using copy() method

// Importing required classes
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

// Main class
// CopyOneCollectionToAnother
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Create destination list
        List<String> destination_List = new ArrayList<>();

        // Add elements
        destination_List.add("Shoes");
        destination_List.add("Toys");
        destination_List.add("Horse");
        destination_List.add("Tiger");

        // Print the elements
        System.out.println(
            "The Original Destination list is ");

        for (int i = 0; i < destination_List.size(); i++) {
            System.out.print(destination_List.get(i) + " ");
        }
```

```
        System.out.println();

        // Create source list
        List<String> source_List = new ArrayList<>();

        // Add elements
        source_List.add("Bat");
        source_List.add("Frog");
        source_List.add("Lion");

        // Copy the elements from source to destination
        Collections.copy(destination_List, source_List);

        // Printing the modified list
        System.out.println(
            "The Destination List After copying is ");

        for (int i = 0; i < destination_List.size(); i++) {
            System.out.print(destination_List.get(i) + " ");
        }
    }
}
```

Output

```
The Original Destination list is
Shoes Toys Horse Tiger
The Destination List After copying is
Bat Frog Lion Tiger
```

Operation 5: Disjoint Collection

java.util.Collections.disjoint() is used to check whether two specified collections are disjoint or not. More formally, two collections are disjoint if they have no elements in common. It returns true if the two collections do not have any element in common.

Example

```
// Java Program to Illustrate Working of Disjoint Function

// Importing required classes
import java.util.ArrayList;
```

```
import java.util.Collections;
import java.util.List;

// Main class
// DisjointCollectionsExample
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Create list1
        List<String> list1 = new ArrayList<>();

        // Add elements to list1
        list1.add("Shoes");
        list1.add("Toys");
        list1.add("Horse");
        list1.add("Tiger");

        // Create list2
        List<String> list2 = new ArrayList<>();

        // Add elements to list2
        list2.add("Bat");
        list2.add("Frog");
        list2.add("Lion");

        // Check if disjoint or not
        System.out.println(
            Collections.disjoint(list1, list2));
    }
}
```

Output

```
true
```