java.util.Collections Class API Guide

java.util.Collections class consists exclusively of static methods that operate on or return collections.

It contains *polymorphic algorithms* that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.
The methods of this class all throw a *NullPointerException* if the collections or class objects provided to them are null.

In this post, we will explore few useful *Collections* class methods with examples

java.util.Collections methods

- *sort(List list)*
- *sort(List list, Comparator<? super Project> c)*
- *shuffle(List<?> list)*
- *reverse(List<?> list)*
- *rotate(List<?> list, int distance)*
- *swap(List<?> list, int i, int j)*
- *replaceAll(List list, String oldVal, String newVal)*
- *copy(List<? super String> dest, List<? extends String> src)*
- *Collections.binarySearch(list, "element 4")*
- *frequency(Collection<?> c, Object o)*
- *disjoint(Collection> c1, Collection> c2)*
- *min(Collection extends ?> coll)*
- *max(Collection extends ?> coll)*

java.util.Collections Class

The *Collections* class provides static methods whose first argument is the collection on which the operation is to be performed The great majority of the algorithms provided by the Java platform operate on List instances, but a few of them operate on arbitrary Collection instances. We use Collections class to demonstrate below *Algorithms*with examples:

- Sorting
- Shuffling
- Routine Data Manipulation
- Searching
- Composition
- Finding Extreme Values

Sorting using *java.util.Collections* Class

The *Collections* class provides two sorting methods:

sort(List list)

Sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must implement the Comparable interface.

```
List<String> list = new LinkedList<>();
list.add("element 2");
list.add("element 1");
list.add("element 4");
list.add("element 3");
// Sorts the specified list into ascending order, according to
// the natural ordering of its elements.
Collections.sort(list);
for (String str : list) {
 System.out.println(" sort elements in ascending order  --" + str);
}
```

Output:

```
sort elements in ascending order  --element 1
sort elements in ascending order  --element 2
sort elements in ascending order  --element 3
sort elements in ascending order  --element 4
```

sort(List list, Comparator<? super Project> c)

Sorts the specified list according to the order induced by the specified comparator. All elements in the list must be mutually comparable using the specified comparator (that is, c.compare(e1, e2) must not throw a *ClassCastException* for any elements e1 and e2 in the list). Example:

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

public class CollectionsClassExamples {

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

```
 private static void sortingCustomObjectsByComparator() {
  // Sort Projects by project id in ascending order.
  List<Project> projects = new ArrayList<>();
  Project project = new Project();
  project.setProjectId(100);
  project.setProjectName("TMS");
  projects.add(project);

  Project project2 = new Project();
  project2.setProjectId(200);
  project2.setProjectName("APEX");
  projects.add(project2);

  Project project3 = new Project();
  project3.setProjectId(50);
  project3.setProjectName("CMS");
  projects.add(project3);

  // Sorting project by project name in ascending order in Java
  Collections.sort(projects, Comparator.comparing(Project::getProjectName));
  printList(projects);

 }

 private static void printList(List<Project> projects) {
  for (Project project : projects) {
   System.out.println(project.getProjectId());
   System.out.println(project.getProjectName());
  }
 }

}

class Project implements Comparable<Project> {
 private int projectId;
 private String projectName;

 public int getProjectId() {
  return projectId;
 }
```

```
  public void setProjectId(int projectId) {
   this.projectId = projectId;
  }

  public String getProjectName() {
   return projectName;
  }

  public void setProjectName(String projectName) {
   this.projectName = projectName;
  }

  @Override
  public int compareTo(Project project) {
   return this.getProjectId() - project.getProjectId();
  }
 }
```

Output:

```
200
APEX
50
CMS
100
TMS
```

Shuffling using *java.util.Collections* Class

The shuffle algorithm does the opposite of what sort does, destroying any trace of order that may have been present in a List. That is, this algorithm reorders the List based on input from a source of randomness such that all possible permutations occur with equal likelihood, assuming a fair source of randomness. Collections utility class provides the shuffling methods.

shuffle(List<?> list)

Randomly permutes the specified list using a default source of randomness. All permutations occur with approximately equal likelihood.
Example:

```
 private static void shuffleAlgorithmsDemo() {
  List<String> list = new LinkedList<>();
  list.add("element 2");
```

```
  list.add("element 1");
  list.add("element 4");
  list.add("element 3");

  Collections.sort(list);
  for (String str : list) {
   System.out.println(" sort elements in ascending order  --" + str);
  }

  // randomly permutes the elements in a List.
  Collections.shuffle(list);
  for (String str : list) {
   System.out.println(" sort elements in ascending order  --" + str);
  }
 }
```

Output:

```
sort elements in ascending order  --element 2
sort elements in ascending order  --element 3
sort elements in ascending order  --element 4
shuffle elements  --element 1
shuffle elements  --element 3
shuffle elements  --element 4
shuffle elements  --element 2
```

Routine Data Manipulation

The Collections class provides five algorithms for doing routine data manipulation on List objects, all of which are pretty straightforward:

- *reverse* - reverses the order of the elements in a List.
- fill - overwrites every element in a List with the specified value. This operation is useful for reinitializing a List.
- *copy* - takes two arguments, a destination *List*, and a source List, and copies the elements of the source into the destination, overwriting its contents. The destination List must be at least as long as the source. If it is longer, the remaining elements in the destination List are unaffected.
- *swap* - swaps the elements at the specified positions in a List.
- *addAll* - adds all the specified elements to a Collection. The elements to be added may be specified individually or as an array.

reverse(List<?> list)

Reverses the order of the elements in the specified list.
Example:

```
List<String> list = new LinkedList<>();
list.add("element 2");
list.add("element 1");
list.add("element 4");
list.add("element 3");

Collections.sort(list);
for(String str : list){
  System.out.println(" sort elements in ascending order  --" + str);
}
//reverses the order of the elements in a List.
Collections.reverse(list);
```

rotate(List<?> list, int distance)

Rotates the elements in the specified list by the specified distance. After calling this method, the element at index i will be the element previously at index (i - distance) mod list.size(), for all values of i between 0 and list.size()-1, inclusive. (This method has no effect on the size of the list.)
Example:

```
List<String> list = new LinkedList<>();
list.add("element 2");
list.add("element 1");
list.add("element 4");
list.add("element 3");

// rotates all the elements in a List by a specified distance.
Collections.rotate(list, 1);
```

swap(List<?> list, int i, int j)

Swaps the elements at the specified positions in the specified list. (If the specified positions are equal, invoking this method leaves the list unchanged.)
Example:

```
List<String> list = new LinkedList<>();
list.add("element 2");
list.add("element 1");
```

```
list.add("element 4");
list.add("element 3");

//swaps the elements at specified positions in a List.
Collections.swap(list, 0, 1);
printMessage(list, "swap elements ");
```

replaceAll(List list, String oldVal, String newVal)

Replaces all occurrences of one specified value in a list with another. More formally, replaces with newVal each element e in list such that (oldVal==null ? e==null : oldVal.equals(e)). (This method has no effect on the size of the list.)
Example:

```
List<String> list = new LinkedList<>();
list.add("element 2");
list.add("element 1");
list.add("element 4");
list.add("element 3");

//replaces all occurrences of one specified value with another.
Collections.replaceAll(list, "element 3", "element 6");
```

copy(List<? super String> dest, List<? extends String> src)

Copies all of 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:

```
List<String> list = new LinkedList<>();
list.add("element 2");
list.add("element 1");
list.add("element 4");
list.add("element 3");

List<String> destList = new ArrayList<>();
Collections.copy(destList, list);
```

Searching

The *binarySearch* algorithm searches for a specified element in a sorted List. This algorithm has two forms.
The first takes a *List* and an element to search for (the "search key"). This form assumes that the List is sorted in ascending order according to the natural ordering of its elements.
The second form takes a Comparator in addition to the List and the search key and assumes that the List is sorted into ascending order according to the specified Comparator. The sort algorithm can be used to sort the List prior to calling *binarySearch*. Example:

```
 private static void searchingAlgorithmsDemo() {
  List<String> list = new LinkedList<>();
  list.add("element 2");
  list.add("element 1");
  list.add("element 4");
  list.add("element 3");

  Collections.sort(list);
  for (String str : list) {
   System.out.println(" sort elements in ascending order  --" + str);
  }
  int index = Collections.binarySearch(list, "element 4");
  System.out.println("Element found at ::" + index);
 }
```

Output:

```
 sort elements in ascending order  --element 1
 sort elements in ascending order  --element 2
 sort elements in ascending order  --element 3
 sort elements in ascending order  --element 4
Element found at ::3
```

Composition

The frequency and disjoint algorithms test some aspect of the composition of one or more Collections:

- *frequency* - counts the number of times the specified element occurs in the specified collection.
- *disjoint* - determines whether two Collections are disjoint; that is, whether they contain no elements in common.

frequency(Collection<?> c, Object o)

Returns the number of elements in the specified collection equal to the specified object. More formally, returns the number of elements e in the collection such that (o == null ? e == null : o.equals(e)).
Example:

```
List<String> list = new LinkedList<>();
list.add("element 2");
list.add("element 1");
list.add("element 1");
list.add("element 3");
//Returns the number of elements in the specified collection
//equal to the specified object.
System.out.println(Collections.frequency(list, "element 1"));
```

disjoint(Collection> c1, Collection> c2)

Returns true if the two specified collections have no elements in common.
Example:

```
List<String> list = new LinkedList<>();
list.add("element 2");
list.add("element 1");
list.add("element 1");
list.add("element 3");
//Returns the number of elements in the specified collection
//equal to the specified object.

System.out.println(Collections.frequency(list, "element 1"));
List<String> list2 = new LinkedList<>();
list2.add("element 2");
list2.add("element 1");
list2.add("element 1");
list2.add("element 3");
//Returns true if the two specified collections have no elements in common.
System.out.println(Collections.disjoint(list, list2));
```

Finding Extreme Values(*min* and *max* methods)

The *min* and the *max* algorithms return, respectively, the minimum and maximum element contained in a specified Collection. Both of these operations come in two forms.

The simple form takes only a Collection and returns the minimum (or maximum) element according to the elements' natural ordering.
The second form takes a Comparator in addition to the Collection and returns the minimum (or maximum) element according to the specified Comparator.

min(Collection extends ?> coll)

Returns the minimum element of the given collection, according to the natural ordering of its elements. All elements in the collection must implement the Comparable interface.
Furthermore, all elements in the collection must be mutually comparable (that is, e1.compareTo(e2) must not throw a *ClassCastException* for any elements e1 and e2 in the collection). Example:

```
List<Integer> list = new LinkedList<>();
list.add(100);
list.add(300);
list.add(200);
list.add(500);
// Returns the minimum element of the given collection,
// according to the natural ordering of its elements.
// All elements in the collection must implement the Comparable interface.
System.out.println(Collections.min(list));
```

max(Collection extends ?> coll)

Returns the maximum element of the given collection, according to the natural ordering of its elements. All elements in the collection must implement the Comparable interface.
Furthermore, all elements in the collection must be mutually comparable (that is, e1.compareTo(e2) must not throw a *ClassCastException* for any elements e1 and e2 in the collection).
Example:

```
List<Integer> list = new LinkedList<>();
list.add(100);
list.add(300);
list.add(200);
list.add(500);
//All elements in the collection must implement the Comparable interface.
System.out.println(Collections.max(list));
```