Java → Collections → The utility class Collections

Theory: The utility class Collections

28 minutes 0 / 5 problems solved

Skip this topic Start practicing

1016 users solved this topic. Latest completion was about 2 hours ago.

The *Java Collections Framework* includes the utility class `Collections`, that contains a number of static methods for creating and processing collections. Some of the methods represent generic algorithms, which means they can work with different types of collections.

It is often the case that programmers forget about this class and reinvent its methods from scratch. Obviously, it's better to remember about this class and check whether it contains the operations you need to perform with a collection.

Let's consider some groups of the provided methods. The full list of the method is available in the official documentation.

> Please, do not confuse the `Collections` class and the `Collection` interface. They both belong to the `java.util` package but represent completely different things.

§1. Creating immutable collections

The first group is a set of methods for creating empty and single-element **immutable** collections.

```
List<String> emptyList = Collections.emptyList();
Set<Integer> emptySet = Collections.emptySet();

List<Integer> singletonList = Collections.singletonList(100);
Set<String> singletonSet = Collections.singleton("Hello");
```

Using these methods look pretty straightforward. But why do we need empty and single element collections? For example, empty collections are often used as the return values from methods instead of `null` to avoid NPE.

```
public static Collection<Integer> algorithm(Collection<Integer> numbers) {
    // lots lines of codes
    if (some_condition) {
        return Collections.emptyList(); // instead of null
    }
    // lots lines of codes
}
```

Singleton collections are extremely optimized to work with a single value. As an example, the class `SingletonList<E>` looks like this:

```
class SingletonList<E> extends .. implements ... {

    private final E element;  // storing a single elment

    SingletonList(E obj) {
        element = obj;
    }

    // some fields and methods

}
```

Apart from this, the class also provides methods to create immutable collections from other collections:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(10);
numbers.add(12);
List<Integer> immutableList = Collections.unmodifiableList(numbers);
```

There are similar methods: `unmodifiableSet(set)` and more.

Current topic:

The utility class Collections ···

Topic depends on:

× List ···

Table of contents:

Remember that it's impossible to change elements within immutable collections. Methods that change elements (`add`, `clear` and so on) will throw `UnsupportedOperationException` when being invoked.

```
List<Integer> singletonList = Collections.singletonList(10);
singletonList.add(20); // throws UnsupportedOperationException
```

Starting with Java 9, there is an alternative way to create immutable collections: `List.of()`, `List.of(1, 2)`, `Set.of("Hello")`. But it is still useful to know about the previous way of doing that since it is often present in existing code.

We skipped the methods for creating maps, but they look very similar. If you need them, just look into the documentation.

§2. Processing lists

There are also some methods for performing list-specific operations: **sorting**, **reversing**, **rotating**, and **shuffling** lists.

Check them out on the following example:

```
var numbers = new ArrayList<>(List.of(1, 2, 3, 2, 3, 4)); // getting a mutable list

Collections.sort(numbers);    // [1, 2, 2, 3, 3, 4]
Collections.reverse(numbers); // [4, 3, 3, 2, 2, 1]
Collections.shuffle(numbers); // randomly permutes the list

System.out.println(numbers);  // a result can be any: [4, 2, 3, 2, 3, 1]
```

The `rotate` method shifts the elements in the specified list by the given distance.

```
List<Integer> numbers = new ArrayList<>(List.of(1, 2, 3, 2, 3, 4));

Collections.rotate(numbers, 1); // [4, 1, 2, 3, 2, 3]
Collections.rotate(numbers, 2); // [2, 3, 4, 1, 2, 3]
```

These methods can be very useful in many applications. The listed methods have overloaded versions as well.

§3. Calculations on collections

There are some methods that can be applied to any collections since the methods take the `Collection` interface as the argument.

- `frequency` counts the number of elements equal to the specified object;
- `min` and `max` according to the natural order of elements;
- `disjoint` checks the two collections do not contain common elements.

Here is an example of applying the listed methods.

```
List<Integer> numbers = List.of(1, 2, 3, 2, 3, 4);

System.out.println(Collections.frequency(numbers, 3)); // 2
System.out.println(Collections.min(numbers)); // 1
System.out.println(Collections.max(numbers)); // 4

System.out.println(Collections.disjoint(numbers, List.of(1, 2))); // false
System.out.println(Collections.disjoint(numbers, List.of(5, 6))); // true
```

If the collection is empty, the methods finding **min** and **max** will throw `NoSuchElementException`. But the `frequency` will just return 0.

The `Collections` class contains some other methods for working with collections as well.

§4. A tricky example

We would like to demonstrate one tricky and interesting example with some modifying operations on **immutable** collections. Just take a look at the following code:

```
List<Integer> singletonList = Collections.singletonList(1);

Collections.sort(singletonList);    // it doesn't throw an exception
Collections.shuffle(singletonList); // it doesn't throw an exception

List<Integer> numbers = Collections.unmodifiableList(List.of(2, 1, 3));
Collections.shuffle(numbers); // it throws UnsupportedOperationException
```

The first and second operations work without throwing an exception since a list containing only a single element doesn't require any modifications to be sorted or shuffled unlike the list with three elements. But if you replace `Collections.singletonList(1)` with `List.of(1)`, the first and second operations will also fail. Even immutable collections have behavioral peculiarities.

> In order not to confuse other programmers it's better not to rely on such somewhat counterintuitive features of Java in your solutions, even if they are fun enough. After a while, you will also forget why such code works.

§5. Conclusion

We've considered the `Collections` class that provides a set of useful methods for collections. Before you start writing your code when working on a problem related to processing collections it is a good idea to check the suitable methods in this class. It will allow you not to reinvent a wheel and use standard methods achieving good performance.