Creating and Processing Data with the Collections Factory Methods

Creating Immutable Collections

Java SE 9 saw the addition of a set of factory methods to the `List` and `Set` interfaces to create lists and sets. The pattern is very simple: just call the `List.of()` or `Set.of()` static method, pass the elements of your list and set, and that's it.

```
1 List<String> stringList = List.of("one", "two", "three");
2 Set<String> stringSet = Set.of("one", "two", "three");
```

Several points are worth noting though.

- The implementation you get in return may vary with the number of elements you put in your list or set. None of them is `ArrayList` or `HashSet`, so your code should not rely on that.
- Both the list and the set you get are immutable structures. You cannot add or modify elements in them, and you cannot modify these elements. If the objects of these structures are mutable, you can still mutate them.
- These structures do not accept null values. If you try to add a `null` value in such a list or set, you will get an exception.
- The `Set` interface does not allow duplicates: this is what a set is about. Because it would not make sense to create such a set with duplicate values, it is assumed that writing such a code is a bug. So you will get an exception if you try to do that.
- The implementations you get are `Serializable`.

These `of()` methods are commonly referred to as *convenience factory methods for collections*.

Getting an Immutable Copy of a Collection

Following the success of the convenience factory methods for collections, another set of convenience methods have been added in Java SE 10 to create immutable copies of collections.

There are two of them: `List.copyOf()` and `Set.copyOf()`. They both follow the same pattern:

```
Collection<String> strings = Arrays.asList("one", "two", "three");

List<String> list = List.copyOf(strings);
Set<String> set = Set.copyOf(strings);
```

In all cases, the collection you need to copy should not be null and should not contain any null elements. If this collection has duplicates, only one of these elements will be kept in the case of `Set.copyOf()`.

What you get in return is an immutable copy of the collection passed as an argument. So modifying this collection will not be reflected in the list or set you get as a copy.

None of the implementations you get accept `null` values. If you try to copy a collection with `null` values, you will get a `NullPointerException`.

Wrapping an Array in a List

The Collections Framework has a class called `Arrays` with about 200 methods to handle arrays. Most of them are implementing various algorithms on arrays, like sorting, merging, searching, and are not covered in this section.

There is one though that is worth mentioning: `Arrays.asList()`. This method takes a vararg as an argument and returns a `List` of the elements you passed, preserving their order. This method is not part of the *convenience factory methods for collections* but is still very useful.

This `List` acts as a wrapper on an array, and behaves in the same way, which maybe a little confusing at first. Once you have set the size of an array, you cannot change it. It means that

you cannot add an element to an existing array, nor can you remove an element from it. All you can do is replace an existing element with another one, possibly null.

The `List` you get by calling `Arrays.asList()` does exactly this.

- If you try to add or remove an element, you will get an `UnsupportedOperationException`, whether you do that directly or through the iterator.
- Replacing existing elements is OK.

So this list is not immutable, but there are restrictions on how you can change it.

Using the Collections Factory Class to Process a Collection

The Collections Framework comes with another factory class: `Collections`, with a set of method to manipulate collections and their content. There are about 70 methods in this class, it would be tedious so examine them one-by-one, so let us present a subset of them.

Extracting the Minimum or the Maximum from a Collection

The `Collections` class give you two methods for that: the `min()` and the `max()`. Both methods take the collection as an argument from which the min or the max is extracted. Both methods have an overload that also takes a comparator as a further argument.

If no comparator is provided then the elements of the collection must implement `Comparable`. If not, a `ClassCastException` will be raised. If a comparator is provided, then it will be used to get the min or the max, whether the elements of the collection are comparable or not.

Getting the min or the max of an empty collection with this method will raise a `NoSuchMethodException`.

Finding a Sublist in a List

Two methods locate a given sublist in a bigger list:

- `indexOfSublist(List<?> source, List<?> target)`: returns the first index of the first element of the `target` list in the `source` list, or -1 if it does not exist;
- `lastIndexOfSublist(List<?> source, List<?> target)`: return the last of these indexes.

Changing the Order of the Elements of a List

Several methods can change the order of the elements of a list:

- `sort()` sorts the list in place. This method may take a comparator as an argument. As usual, if no comparator is provided, then the elements of the list must be comparable. If a comparator is provided, then it will be used to compare the elements. Starting with Java SE 8, you should favor the `sort()` method from the `List`interface.
- `shuffle()` randomly shuffles the elements of the provided list. You can provide yout instance of `Random` if you need a random shuffling that you can repeat.
- `rotate()` rotates the elements of the list. After a rotation the element at index 0 will be found at index 1 and so on. The last elements will be moved to the first place of the list. You can combine `subList()` and `rotate()` to remove an element at a given index and to insert it in another place in the list. This can be done with the following code:

```
List<String> strings = Arrays.asList("0", "1", "2", "3", "4");
System.out.println(strings);
int fromIndex = 1, toIndex = 4;
Collections.rotate(strings.subList(fromIndex, toIndex), -1);
System.out.println(strings);
```

The result is the following:

```
[0, 1, 2, 3, 4]
[0, 2, 3, 1, 4]
```

The element at index `fromIndex` has been removed from its place, the list has been reorganized accordingly, and the element has been inserted at index `toIndex - 1`.

- `reverse()`: reverse the order of the elements of the list.
- `swap()`: swaps two elements from the list. This method can take a list as an argument, as well as a plain array.

Wrapping a Collection in an Immutable Collection

The `Collections` factory class gives you several methods to create immutable wrappers for your collections or maps. The content of the structure is not duplicated; what you get is a wrapper around your structure. All the attempts to modify it will raise exceptions.

All these methods starts with `unmodifiable`, followed by the name of the type of your structure. For instance, to create an immutable wrapper of a list, you can call:

```
List<String> strings = Arrays.asList("0", "1", "2", "3", "4");
List<String> immutableStrings = Collections.unmodifiableList(strings);
```

Just a word of warning: it is not possible to modify your collection through this wrapper. But this wrapper is backed by your collection, so if you modify it by another means, this modification will be reflected in the immutable collection. Let us see that in the following code:

```
List<String> strings = new ArrayList<>(Arrays.asList("0", "1", "2", "3", "4"));
List<String> immutableStrings = Collections.unmodifiableList(strings);
System.out.println(immutableStrings);
strings.add("5");
System.out.println(immutableStrings);
```

Running this example will give you the following:

```
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
```

If you plan to create an immutable collection using this pattern, defensively copying it first may be a safe precaution.

Wrapping a Collection in a Synchronized Collection

In the same way as you can create immutable wrappers for your maps and collections, the `Collections` factory class can create synchronized wrappers for them. The patterns follow the same naming convention as the names for methods that create immutable wrappers: the methods are called `synchronized` followed by `Collection`, `List`, `Set`, etc...

There are two precautions you need to follow.

- All the accesses to your collection should be made through the wrapper you get
- Traversing your collection with an iterator or a stream should be synchronized by the calling code on the list itself.

Not following these rules will expose your code to race conditions.

Synchronizing collections using the `Collections` factory methods may not be your best choice. The Java Util Concurrent framework has better solutions to offer.

Last update: *September 14, 2021*