Java 9 Convenience Factory Methods for Collections

Last modified: April 15, 2018

by baeldung (/author/baeldung/)

Java (/category/java/) +

Java 9 (/tag/java-9/) **Java Collections (/tag/collections/)**

1. Overview

Java 9 brings the long-awaited syntactic sugar for creating small unmodifiable *Collection* instance*s* using a concise one line code. As per JEP 269 (http://openjdk.java.net/jeps/269), new convenience factory methods will be included in JDK 9.

In this article, we'll cover its usage along with the implementation details.

2. History and Motivation

Creating a small immutable *Collection* in Java is very verbose using the traditional way.

Let's take an example of a *Set*:

```java
Set<String> set = new HashSet<>();
set.add("foo");
set.add("bar");
set.add("baz");
set = Collections.unmodifiableSet(set);
```

That's way too much code for a simple task and it should be possible to be done in a single expression.

The is also true for a *Map*, however, for *List*, there's a factory method:

```java
List<String> list = Arrays.asList("foo", "bar", "baz");
```

Although this *List* creation is better than the constructor initialization, this is less obvious (https://en.wikipedia.org/wiki/Principle_of_least_astonishment) as the common intuition would not be to look into *Arrays* class for methods to create a *List*:

There are other ways of reducing verbosity like the **double-brace** technique:

```java
Set<String> set = Collections.unmodifiableSet(new HashSet<String>() {{
    add("foo"); add("bar"); add("baz");
}});
```

or by using Java 8 *Streams*:

```java
Stream.of("foo", "bar", "baz")
  .collect(collectingAndThen(toSet(), Collections::unmodifiableSet));
```

The double brace technique is only a little less verbose but greatly reduces the readability (and is considered an anti-pattern).

The Java 8 version, though, is a one-line expression, has some problems too. First, it's not obvious and intuitive, second, it's still verbose, third, it involves the creation of unnecessary objects and fourth, this method can't be used for creating a *Map*.

To summarize the shortcomings, none of the above approaches treat the specific use case creating a small unmodifiable *Collection* first-class class problem.

3. Description and Usage

Static methods have been provided for *List*, *Set*, and *Map* interfaces which take the elements as arguments and return an instance of *List*, *Set* and *Map* respectively.

This method is named *of(…)* for all the three interfaces.

3.1. *List* and *Set*

The signature and characteristics of *List* and *Set* factory methods are same:

```java
static <E> List<E> of(E e1, E e2, E e3)
static <E> Set<E>  of(E e1, E e2, E e3)
```

usage of the methods:

```java
List<String> list = List.of("foo", "bar", "baz");
Set<String> set = Set.of("foo", "bar", "baz");
```

As you can see, it's very simple, short and concise.

In the example, we've used the method with takes exactly three elements as parameters and returns a *List* / *Set* of size 3.

But, there are 12 overloaded versions of this method – eleven with 0 to 10 parameters and one with var-args:

```java
static <E> List<E> of()
static <E> List<E> of(E e1)
static <E> List<E> of(E e1, E e2)
// ....and so on

static <E> List<E> of(E... elems)
```

For most practical purposes, 10 elements would be sufficient but if more are required, the var-args version can be used.

Now you may ask, what is the point of having 11 extra methods if there's a var-args version which can work for any number of elements.

The answer to that is performance. **Every var-args method call implicitly creates an array. Having the overloaded methods avoid unnecessary object creation and the garbage collection overhead thereof.**

During the creation of a *Set* using a factory method, if duplicate elements are passed as parameters, then *IllegalArgumentException* is thrown at runtime:

```
@Test(expected = IllegalArgumentException.class)
public void onDuplicateElem_IfIllegalArgExp_thenSuccess() {
    Set.of("foo", "bar", "baz", "foo");
}
```

An important point to note here is that since the factory methods use generics, primitive types get autoboxed.

If an array of primitive type is passed, a *List* of *array* of that primitive type is returned.

For example:

```
int[] arr = { 1, 2, 3, 4, 5 };
List<int[]> list = List.of(arr);
```

In this case, a *List<int[]>* of size 1 is returned and the element at index 0 contains the array.

3.2. *Map*

The signature of *Map* factory method is:

```
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3)
```

and the usage:

```
Map<String, String> map = Map.of("foo", "a", "bar", "b", "baz", "c");
```

Similarly to *List* and *Set*, the *of(...)* method is overloaded to have 0 to 10 key-value pairs.

In the case of *Map*, there is a different method for more than 10 key-value pairs:

```java
static <K,V> Map<K,V> ofEntries(Map.Entry<? extends K,? extends V>...
```

and it's usage:

```java
Map<String, String> map = Map.ofEntries(
  new AbstractMap.SimpleEntry<>("foo", "a"),
  new AbstractMap.SimpleEntry<>("bar", "b"),
  new AbstractMap.SimpleEntry<>("baz", "c"));
```

Passing in duplicate values for Key would throw an *IllegalArgumentException*:

```java
@Test(expected = IllegalArgumentException.class)
public void givenDuplicateKeys_ifIllegalArgExp_thenSuccess() {
    Map.of("foo", "a", "foo", "b");
}
```

Again, in the case of *Map* too, the primitive types are autoboxed.

4. Implementation Notes

The collections created using the factory methods are not commonly used implementations.

For example, the *List* is not an *ArrayList* and the *Map* is not a *HashMap*. Those are different implementations which are introduced in Java 9. These implementations are internal and their constructors have restricted access.

In this section, we'll see some important implementation differences which are common to all the three types of collections.

4.1. Immutable

The collections created using factory methods are immutable and changing an element, adding new elements or removing an element throws *UnsupportedOperationException*:

```java
@Test(expected = UnsupportedOperationException.class)
public void onElemAdd_ifUnSupportedOpExpnThrown_thenSuccess() {
    Set<String> set = Set.of("foo", "bar");
    set.add("baz");
}
```

```java
@Test(expected = UnsupportedOperationException.class)
public void onElemModify_ifUnSupportedOpExpnThrown_thenSuccess() {
    List<String> list = List.of("foo", "bar");
    list.set(0, "baz");
}
```

```java
@Test(expected = UnsupportedOperationException.class)
public void onElemRemove_ifUnSupportedOpExpnThrown_thenSuccess() {
    Map<String, String> map = Map.of("foo", "a", "bar", "b");
    map.remove("foo");
}
```

4.2. No *null* Element Allowed

In the case of *List* and *Set*, no elements can be *null*. In the case of a *Map*, neither keys nor values can be *null*. Passing *null* argument throws a *NullPointerException*:

```java
@Test(expected = NullPointerException.class)
public void onNullElem_ifNullPtrExpnThrown_thenSuccess() {
    List.of("foo", "bar", null);
}
```

4.3. Value-Based Instances

The instances created by factory methods are value based. This means that factories are free to create a new instance or return an existing instance.

Hence, if we create Lists with same values, they may or may not refer to the same object on the heap:

```java
List<String> list1 = List.of("foo", "bar");
List<String> list2 = List.of("foo", "bar");
```

In this case, *list1 == list2* may or may not evaluate to *true* depending on the JVM.

4.4. Serialization

Collections created from factory methods are *Serializable* if the elements of the collection are *Serializable*.

5. Conclusion

In this article, we introduced the new factory methods for Collections introduced in Java 9.

We concluded why this feature is a welcome change by going over some past methods for creating unmodifiable collections. We covered it's usage and highlighted key points to be considered while using them.

Finally, we clarified that these collections are different from the commonly used implementations and pointed out key differences.

The complete source code for this article and the unit tests are available over on GitHub (https://github.com/eugenp/tutorials/tree/master/core-java-9).

▲ newest ▲ oldest ▲ most voted

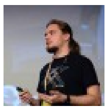
Guest

Vegard Skjefstad (https://www.vegard.net/)

Thanks for the post, to the point and easy to understand.

Minor typo in 4.2, NullPoimterException.

+ 0 − 1 year ago



Guest

Grzegorz Piwowarek Thanks, fixed 🙂

+ 0 −

1 year ago

CATEGORIES

SPRING (/CATEGORY/SPRING/)
REST (/CATEGORY/REST/)
JAVA (/CATEGORY/JAVA/)
SECURITY (/CATEGORY/SECURITY-2/)
PERSISTENCE (/CATEGORY/PERSISTENCE/)
JACKSON (/CATEGORY/JACKSON/)
HTTPCLIENT (/CATEGORY/HTTP/)
KOTLIN (/CATEGORY/KOTLIN/)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (HTTPS://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTPS://WWW.BAELDUNG.COM/JACKSON)
HTTPCLIENT 4 TUTORIAL (HTTPS://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES/)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES/)
SECURITY WITH SPRING (HTTPS://WWW.BAELDUNG.COM/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT/)
THE COURSES (HTTPS://COURSES.BAELDUNG.COM)
CONSULTING WORK (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (/FULL_ARCHIVE)
WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)
CONTACT (/CONTACT)
EDITORS (/EDITORS)