16.14 Java SE 9: Convenience Factory Methods for Immutable Collections[1]

1. If any changes occur to the new Java SE 9 content in this section, we'll post updates on the book's website at `http://www.deitel.com/books/jhtp11`. This example requires JDK 9 to execute.

9

Java SE 9 adds new `static` *convenience factory methods* to interfaces `List`, `Set` and `Map` that enable you to create small *immutable* collections—they cannot be modified once they are created (JEP 269). We introduced factory methods in Section 10.12—the word *factory* indicates that these methods create objects. They're *convenient* because you simply pass the elements as arguments to a convenience factory method, which creates the collection and adds the elements to the collection for you.

The collections returned by the unmodifiable wrappers we discussed in Section 16.12 create *immutable views* of *mutable collections*—the reference to the original mutable collection can still be used to modify the collection. The convenience factory methods instead return custom collection objects that are truly immutable and optimized to store small collections.

In Chapters 17 and 23, we explain how using lambdas and streams with immutable entities can help you create “parallelizable” code that will run more efficiently on today’s multi-core architectures. Figure 16.20 demonstrates these convenience factory methods for a `List`, a `Set` and two `Map`s.

Common Programming Error 16.3

Calling any method that attempts to modify a collection returned by the `List`, `Set` *or* `Map` *convenience factory methods results in an* `UnsupportedOperationException`.

Software Engineering Observation 16.6

In Java, collection elements are always references to objects. The objects referenced by an immutable collection may still be mutable.

```
   1   // Fig. 16.20: FactoryMethods.java
 2   // Java SE 9 collection factory methods.
       3   import java.util.List;
```

```
4   import java.util.Map;
5   import java.util.Set;
6
7   public class FactoryMethods {
8       public static void main(String[] args) {
9           // create a List
10          List<String> colorList = List.of("red", "o
11             "green", "blue", "indigo", "violet");
12          System.out.printf("colorList: %s%n%n", col
13
14          // create a Set
15          Set<String> colorSet = Set.of("red", "oran
16             Set<String> colorSet = Set.of("red", "o
17          System.out.printf("colorSet: %s%n%n", colo
18
19          // create a Map using method "of"
20          Map<String, Integer> dayMap = Map.of("Mond
21             "Wednesday", 3, "Thursday", 4, "Friday"
22             "Sunday", 7);
23          System.out.printf("dayMap: %s%n%n", dayMap
24
25          // create a Map using method "ofEntries" f
26          Map<String, Integer> daysPerMonthMap = Map
27             Map.entry("January", 31),
28             Map.entry("February", 28),
29             Map.entry("March", 31),
30             Map.entry("April", 30),
31             Map.entry("May", 31),
32             Map.entry("June", 30),
33             Map.entry("July", 31),
34             Map.entry("August", 31),
35             Map.entry("September", 30),
36             Map.entry("October", 31),
37             Map.entry("November", 30),
38             Map.entry("December", 31)
39          );
40          System.out.printf("monthMap: %s%n", daysPe
41      }
42  }
```

```
colorList: [red, orange, yellow, green, blue, indigo,

colorSet: [yellow, green, red, blue, violet, indigo,

dayMap: {Tuesday=2, Wednesday=3, Friday=5, Thursday=4
                    Sunday=7}

monthMap: {April=30, February=28, September=30, July=
November=30, December=31, March=31, January=31, June=
```

```
colorList: [red, orange, yellow, green, blue, indigo,

colorSet: [violet, yellow, orange, green, blue, red,

dayMap: {Saturday=6, Tuesday=2, Wednesday=3, Sunday=7
                    Friday=5}

monthMap: {February=28, August=31, July=31, November=
December=31, September=30, January=31, March=31, June
```

Fig. 16.20

Java SE 9 collection factory methods.

`List` Interface's Convenience Factory Method `of`

Lines 10–11 use the `List` convenience factory method `of` to create an immutable `List<String>`. Method `of` has overloads for `List`s of zero to 10 elements and an additional overload that can receive any number of elements. Line 12 displays the `String` representation of the `List`'s contents—recall that this automatically iterates through the `List`'s elements to create the `String`. Also, the returned `List`'s elements are guaranteed to be in the *same* order as method `of`'s arguments.

Performance Tip 16.2

The collections returned by the convenience factory methods are optimized for up to 10 elements (for `List`*s and* `Set`*s) or key–value pairs (for* `Map`*s).*

Software Engineering Observation 16.7

Method `of` *is overloaded for zero to 10 elements because research showed that these handle the vast majority of cases in which immutable collections are needed.*

Performance Tip 16.3

Method `of`*'s overloads for zero to 10 elements eliminate the extra overhead of processing variable-length argument lists. This improves the performance of applications that create small immutable collections.*

Common Programming Error 16.4

The collections returned by the convenience factory methods are not allowed to contain `null` *values—these methods throw a* `NullPointerException` *if any argument is* `null`.

`Set` Interface's Convenience Factory Method `of`

Lines 15–16 use the `Set` convenience factory method `of` to create an immutable `Set<String>`. As with `List`'s method `of`, `Set`'s method `of` has overloads for `Set`s of zero to 10 elements and an additional overload that can receive any number of elements. Line 17 displays the `String` representation of the `Set`'s contents. Note that we showed two

sample outputs of this program and that the order of the `Set`'s elements is *different* in each output. According to the `Set` interface's documentation, the iteration order is *unspecified* for `Set`s returned by the convenience factory methods—as the outputs show, that order can change between executions.

 Common Programming Error 16.5

`Set`*'s method* `of` *throws an* `IllegalArgumentException` *if any of its arguments are duplicates.*

`Map` Interface's Convenience Factory Method `of`

Lines 20–22 use `Map`'s convenience factory method `of` to create an immutable `Map<String, Integer>`. As with `List` and `Set`, `Map`'s method `of` has overloads for `Map`s of zero to 10 key–value pairs. Each pair of arguments (for example, `"Monday"` and `1` in line 20) represents one key–value pair. For `Map`s with more than 10 key–value pairs, interface `Map` provides the method `ofEntries` (which we discuss momentarily). Line 23 displays the `String`

representation of the `Map`'s contents. According to the `Map` interface's documentation, the iteration order is *unspecified* for the keys in `Map`s returned by the convenience factory methods—as the outputs show, that order can change between program executions.

Common Programming Error 16.6

`Map`'s methods `of` and `ofEntries` each throw an `IllegalArgumentException` if any of the keys are duplicates.

`Map` Interface's Convenience Factory Method `ofEntries`

Lines 26–39 use the `Map` convenience factory method `ofEntries` to create an immutable `Map<String, Integer>`. Each of this method's variable number of arguments is the result of a call to `Map`'s `static` method `entry`, which creates and returns a `Map.Entry` object representing one key–value pair. Line 40 displays the `String` representation of the `Map`'s contents. The outputs confirm once again that the iteration order of a `Map`'s keys can change

between program executions.