Part 3 Effective Programming with Streams and Lambdas

8 Collection API enhancements

This chapter covers

- Using collection factories
- Learning new idiomatic patterns to use with List and Set
- Learning idiomatic patterns to work with Map

8.1 Collection factories

- Java 9 introduced a few convenient ways to create small collection objects.
- `Arrays.asList()`

```
List<String> friends
      = Arrays.asList("Raphael", "Olivia", "Thibaut");
```

// You can get a fixed-sized list

```
List<String> friends = Arrays.asList("Raphael", "Olivia");
friends.set(0, "Richard");
friends.add("Thibaut");
```

throws an UnsupportedOperationException

- Set and Map

- Set

```
Set<String> friends "
    = new HashSet<>(Arrays.asList("Raphael", "Olivia",
Thibaut"));


Set<String> friends
     = Stream.of("Raphael", "Olivia", "Thibaut")
             .collect(Collectors.toSet());
```

- You get a mutable Set as a result.

List factory

- `List.of()` // an immutable list

```
List<String> friends = List.of("Raphael", "Olivia", "Thibaut");
System.out.println(friends);
```

[Raphael, Olivia, Thibaut]

- `Collectors.toList()` // transform a stream to list

Set factory

- `Set.of()`

```
Set<String> friends = Set.of("Raphael", "Olivia", "Thibaut");
System.out.println(friends);
```

[Raphael, Olivia, Thibaut]

```
Set<String> friends = Set.of("Raphael", "Olivia", "Olivia");
```

java.lang.IllegalArgumentException: duplicate element: Olivia

Map factories

- `Map.of()`

```
Map<String, Integer> ageOfFriends
    = Map.of("Raphael", 30, "Olivia", 25, "Thibaut", 26);
System.out.println(ageOfFriends);
```

{Olivia=25, Raphael=30, Thibaut=26}

- `Map.ofEntries()` // takes `Map.Entry<K, V>` objects

```
import static java.util.Map.entry;
Map<String, Integer> ageOfFriends
        = Map.ofEntries(entry("Raphael", 30),
                        entry("Olivia", 25),
                        entry("Thibaut", 26));
System.out.println(ageOfFriends);
```

{Olivia=25, Raphael=30, Thibaut=26}

- `Map.entry` is a new factory method to create `Map.Entry` objects.

8.2 Working with List and Set

- Java 8 introduced a couple of methods into the `List` and `Set` interfaces:
  - `removeIf` removes element matching a predicate. It's available on all classes that implement `List` or `Set` (and is inherited from the Collection interface).
  - `replaceAll` is available on `List` and replaces elements using a (`UnaryOperator`) function.
  - `sort` is also available on the `List` interface and sorts the list itself

removeif

- Consider the following code, which tries to remove transactions that have a reference code starting with a digit:

```
for (Transaction transaction : transactions) {
    if(Character.isDigit(transaction.getReferenceCode().charAt(0))) {
        transactions.remove(transaction);
    }
}
```

- Unfortunately, this code may result in a **`ConcurrentModificationException`**.

- To solve this problem, you have to use the `Iterator` object explicitly and call its `remove()` method:

```
for (Iterator<Transaction> iterator = transactions.iterator();
     iterator.hasNext(); ) {
    Transaction transaction = iterator.next();
    if(Character.isDigit(transaction.getReferenceCode().charAt(0))) {
        iterator.remove();
    }
}
```

- The Java 8 `removeIf` method, which is not only simpler but also protects you from these bugs.

```
transactions.removeIf(transaction ->
    Character.isDigit(transaction.getReferenceCode().charAt(0)));
```

```
for (Iterator<Transaction> iterator = transactions.iterator();
     iterator.hasNext(); ) {
   Transaction transaction = iterator.next();
   if(Character.isDigit(transaction.getReferenceCode().charAt(0))) {
       transactions.remove(transaction);
   }
}
```

Problem we are iterating and modifying the collection through two separate objects

Notice that two separate objects manage the collection:

- The `Iterator` object, which is querying the source by using `next()` and `hasNext()`
- The `Collection` object itself, which is removing the element by calling `remove()`

replaceAll

- The `replaceAll` method on the `List` interface lets you replace each element in a list with a new one.

```
referenceCodes.stream()
              .map(code -> Character.toUpperCase(code.charAt(0)) +
      code.substring(1))
              .collect(Collectors.toList())
              .forEach(System.out::println);
```

[a12, C14, b13]

outputs A12, C14, B13

```
for (ListIterator<String> iterator = referenceCodes.listIterator();
     iterator.hasNext(); ) {
   String code = iterator.next();
   iterator.set(Character.toUpperCase(code.charAt(0)) + code.substring(1));
}
```

```
referenceCodes.replaceAll(code -> Character.toUpperCase(code.charAt(0)) +
      code.substring(1));
```

8.3 Working with Map

- Iterator of a `Map.Entry<K, V>`

```
for(Map.Entry<String, Integer> entry: ageOfFriends.entrySet()) {
    String friend = entry.getKey();
    Integer age = entry.getValue();
    System.out.println(friend + " is " + age + " years old");
}
```

- `forEach` method accepts a `BiConsumer`, taking the key and value as arguments.

```
ageOfFriends.forEach((friend, age) -> System.out.println(friend + " is " +
    age + " years old"));
```

Sorting

- Two new utilities let you sort the entries of a map by values or keys:
  - `Entry.comparingByValue`
  - `Entry.comparingByKey`

```
Map<String, String> favouriteMovies
        = Map.ofEntries(entry("Raphael", "Star Wars"),
        entry("Cristina", "Matrix"),
        entry("Olivia",
        "James Bond"));

favouriteMovies
  .entrySet()
  .stream()
  .sorted(Entry.comparingByKey())
  .forEachOrdered(System.out::println);
```

Processes the elements of the stream in alphabetic order based on the person's name

outputs, in order:

```
Cristina=Matrix
Olivia=James Bond
Raphael=Star Wars
```

`getOfDefault()`

```
Map<String, String> favouriteMovies
        = Map.ofEntries(entry("Raphael", "Star Wars"),
        entry("Olivia", "James Bond"));
System.out.println(favouriteMovies.getOrDefault("Olivia", "Matrix"));
System.out.println(favouriteMovies.getOrDefault("Thibaut", "Matrix"));
```

Outputs Matrix

Outputs James Bond

- Note that if the key existed in the `Map` but was accidentally associated with a null value, `getOrDefault` can still return null.

- Compute patterns
  - Three new operations can help:
    - `computeIfAbsent`—If there's no specified value for the given key (it's absent or its value is null), calculate a new value by using the key and add it to the `Map`.
    - `computeIfPresent`—If the specified key is present, calculate a new value for it and add it to the `Map`.
    - `compute`—This operation calculates a new value for a given key and stores it in the `Map`

- One use of `computeIfAbsent` is for caching information. Suppose that you parse each line of a set of files and calculate their SHA-256 representation. If you've processed the data previously, there's no need to recalculate it.

Now suppose that you implement a cache by using a `Map`, and you use an instance of `MessageDigest` to calculate SHA-256 hashes:

```
Map<String, byte[]> dataToHash = new HashMap<>();
MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");
```

Then you can iterate through the data and cache the results:

```
lines.forEach(line ->
   dataToHash.computeIfAbsent(line,
                              this::calculateDigest));

private byte[] calculateDigest(String key) {
   return messageDigest.digest(key.getBytes(StandardCharsets.UTF_8));
}
```

line is the key to look up in the map.

The operation to execute if the key is absent

The helper that will calculate a hash for the given key

- This pattern is also useful for conveniently dealing with maps that store multiple values. If you need to add an element to a Map<K, List<V>>, you need to ensure that the entry has been initialized. This pattern is a verbose one to put in place. Suppose that you want to build up a list of movies for your friend Raphael:

```
String friend = "Raphael";
List<String> movies = friendsToMovies.get(friend);
if(movies == null) {
    movies = new ArrayList<>();
    friendsToMovies.put(friend, movies);
}
movies.add("Star Wars");

System.out.println(friendsToMovies);
```

Check that the list was initialized.

Add the movie.

{Raphael: [Star Wars]}

- `friendsToMovies.computeIfAbsent("Raphael", name -> new ArrayList<>()).add("Star Wars");`
- Note a subtlety: if the function that produces the value returns null, the current mapping is removed from the Map.

Remove patterns

- Since Java 8, an overloaded version removes an entry only if the key is associated with a specific value.

```
String key = "Raphael";
String value = "Jack Reacher 2";
if (favouriteMovies.containsKey(key) &&
      Objects.equals(favouriteMovies.get(key), value)) {
   favouriteMovies.remove(key);
   return true;
}
else {
   return false;
}
```

- `favouriteMovies.remove(key, value);`

Merge

Suppose that you'd like to merge two intermediate Maps, perhaps two separate Maps for two groups of contacts. You can use putAll as follows:

```
Map<String, String> family = Map.ofEntries(
    entry("Teo", "Star Wars"), entry("Cristina", "James Bond"));
Map<String, String> friends = Map.ofEntries(
    entry("Raphael", "Star Wars"));
Map<String, String> everyone = new HashMap<>(family);
everyone.putAll(friends);
System.out.println(everyone);
```

Copies all the entries from friends into everyone

{Cristina=James Bond, Raphael=Star Wars, Teo=Star Wars}

- The new `merge` method takes a `BiFunction` to merge values that have a duplicate key. Suppose that Cristina is in both the family and friends maps but with different associated movies:

```
Map<String, String> family = Map.ofEntries(
    entry("Teo", "Star Wars"), entry("Cristina", "James Bond"));
Map<String, String> friends = Map.ofEntries(
    entry("Raphael", "Star Wars"), entry("Cristina", "Matrix"));
```

- Then you could use the `merge` method in combination with `forEach` to provide a way to deal with the conflict. The following code concatenates the string names of the two movies:

```
Map<String, String> everyone = new HashMap<>(family);
friends.forEach((k, v) ->
   everyone.merge(k, v, (movie1, movie2) -> movie1 + " & " + movie2));
System.out.println(everyone);
```

Given a duplicate key, concatenates the two values

Outputs {Raphael=Star Wars, Cristina=James Bond & Matrix, Teo=Star Wars}

Replacement patterns

- `Map` has two new methods that let you replace the entries inside a `Map`:
  - `replaceAll` — Replaces each entry's value with the result of applying a `BiFunction`. This method works similarly to `replaceAll` on a `List`, which you saw earlier.
  - `Replace` — Lets you replace a value in the `Map` if a key is present. An additional overload replaces the value only if it the key is mapped to a certain value.

You could format all the values in a `Map` as follows:

```
Map<String, String> favouriteMovies = new HashMap<>();
favouriteMovies.put("Raphael", "Star Wars");
favouriteMovies.put("Olivia", "james bond");
favouriteMovies.replaceAll((friend, movie) -> movie.toUpperCase());
System.out.println(favouriteMovies);
```

We have to use a mutable map since we will be using replaceAll

{Olivia=JAMES BOND, Raphael=STAR WARS}

Note that the `merge` method has a fairly complex way to deal with nulls, as specified in the Javadoc:

> *If the specified key is not already associated with a value or is associated with null, [`merge`] associates it with the given non-null value. Otherwise, [`merge`] replaces the associated value with the [result] of the given remapping function, or removes [it] if the result is null.*

You can also use `merge` to implement initialization checks. Suppose that you have a `Map` for recording how many times a movie is watched. You need to check that the key representing the movie is in the map before you can increment its value:

```
Map<String, Long> moviesToCount = new HashMap<>();
String movieName = "JamesBond";
long count = moviesToCount.get(movieName);
if(count == null) {
   moviesToCount.put(movieName, 1);
}
else {
   moviesToCount.put(moviename, count + 1);
}
```

This code can be rewritten as

```
moviesToCount.merge(movieName, 1L, (key, count) -> count + 1L);
```

8.4 Improved ConcurrentHashMap

- The `ConcurrentHashMap` class was introduced to provide a more modern `HashMap`, which is also concurrency friendly. `ConcurrentHashMap` allows concurrent `add` and `update` operations that lock only certain parts of the internal data structure. Thus, read and write operations have improved performance compared with the synchronized `Hashtable` alternative. (Note that the standard `HashMap` is unsynchronized.)

- Reduce and Search
  - `ConcurrentHashMap` supports three new kinds of operations, reminiscent of what you saw with streams:
    - `forEach`—Performs a given action for each (key, value)
    - `reduce`—Combines all (key, value) given a reduction function into a result
    - `search`—Applies a function on each (key, value) until the function produces a non-null result
  - Each kind of operation supports four forms, accepting functions with keys, values, `Map.Entry`, and (key, value) arguments:
    - Operates with keys and values (`forEach, reduce, search`)
    - Operates with keys (`forEachKey, reduceKeys, searchKeys`)
    - Operates with values (`forEachValue, reduceValues, searchValues`)
    - Operates with `Map.Entry` objects (`forEachEntry, reduceEntries, searchEntries`)

- In addition, you need to specify a parallelism threshold for all these operations. The operations execute sequentially if the current size of the map is less than the given threshold. A value of `1` enables maximal parallelism using the common thread pool. A threshold value of `Long.MAX_VALUE` runs the operation on a single thread. You generally should stick to these values unless your software architecture has advanced resource-use optimization.

In this example, you use the `reduceValues` method to find the maximum value in the map:

A ConcurrentHashMap, presumed to be updated to contain several keys and values

```
ConcurrentHashMap<String, Long> map = new ConcurrentHashMap<>();
long parallelismThreshold = 1;
Optional<Integer> maxValue =
   Optional.ofNullable(map.reduceValues(parallelismThreshold, Long::max));
```

- Counting
  - `mappingCount()` returns the number of mappings in the map as al long.
- Set views
  - The `ConcurrentHashMap` class provides a new method called `keySet` that returns a view of the `ConcurrentHashMap` as a `Set`. (Changes in the map are reflected in the `Set`, and vice versa.)
  - You can also create a `Set` backed by a `ConcurrentHashMap` by using the new static method `newKeySet`.