Baeldung

Working with Java Streams eBook

V 1.0.0

1: Introduction to Java 8 Streams

2. Guide to Java 8's Collectors

3. Java Stream Filter With Lambda Expression

4. Working With Maps Using Streams

5. The Difference Between map() and flatMap()

6. When to Use a Parallel Stream in Java

7. Guide to Java 8 groupingBy Collector

8. Guide to Stream.reduce()

1: Introduction to Java 8 Streams

1. Introduction

In this chapter, we'll take a quick look at one of the major pieces of new functionality that Java 8 has added, Streams.

We'll explore what streams are about, and showcase the creation and basic stream operations with simple examples.

One of the major new features in Java 8 is the introduction of the stream functionality, *java.util.stream*, which contains classes for processing sequences of elements.

The central API class is the *Stream<T>*. The following section will demonstrate how streams can be created using the existing data-provider sources.

2.1. Stream Creation

Streams can be created from different element sources, like collections or arrays, with the help of the *stream()* and *of()* methods:

```
1. String[] arr = new String[]{“a”, “b”, “c”};
2. Stream<String> stream = Arrays.stream(arr);
3. stream = Stream.of(“a”, “b”, “c”);
```

A *stream()* default method is added to the *Collection* interface and allows us to create a *Stream<T>* using any collection as an element source:

```
1. Stream<String> stream = list.stream();
```

2.2. Multi-threading With Streams

Stream API also simplifies multithreading by providing the *parallelStream()* method that runs operations over the stream's elements in parallel mode.

The code below allows us to run the *doWork()* method in parallel for every element of the stream:

```
1. list.parallelStream().forEach(element -> doWork(element));
```

In the following section, we'll introduce some of the basic Stream API operations.

3. Stream Operations

There are many useful operations that can be performed on a stream.

They're divided into **intermediate operations** (return *Stream<T>*) and **terminal operations** (return a result of definite type). Intermediate operations allow chaining.

It's also worth noting that operations on streams don't change the source. Here's a quick example:

```
1.  long count = list.stream().distinct().count();
```

So the *distinct()* method represents an intermediate operation, which creates a new stream of unique elements of the previous stream, and the *count()* method is a terminal operation, which returns the stream's size.

3.1. Iterating

Stream API helps to substitute *for*, *for-each*, and *while* loops. It allows us to concentrate on the operation's logic, but not on the iteration over the sequence of elements:

```
1.  for (String string : list) {
2.      if (string.contains("a")) {
3.          return true;
4.      }
```

This code can be changed with just one line of Java 8 code:

```
1.  boolean isExist = list.stream().anyMatch(element -> element.
2.  contains("a"));
```

3.2. Filtering

The *filter()* method allows us to pick a stream of elements that satisfy a predicate.

For example, consider the following list:

```
1.  ArrayList<String> list = new ArrayList<>();
2.  list.add("One");
3.  list.add("OneAndOnly");
4.  list.add("Derek");
5.  list.add("Change");
6.  list.add("factory");
7.  list.add("justBefore");
8.  list.add("Italy");
9.  list.add("Italy");
10. list.add("Thursday");
11. list.add("");
12. list.add("");
```

The following code creates a *Stream<String>* of the *List<String>*, finds all elements of this stream that contain *char "d"*, and creates a new stream containing only the filtered elements:

```
1.  Stream<String> stream = list.stream().filter(element -> element.
2.  contains("d"));
```

3.3. Mapping

To convert elements of a *Stream* by applying a special function to them, and to collect these new elements into a *Stream*, we can use the *map()* method:

```
1.  List<String> uris = new ArrayList<>();
2.  uris.add("C:\\My.txt");
3.  Stream<Path> stream = uris.stream().map(uri -> Paths.get(uri));
```

So the code above converts *Stream<String>* to the *Stream<Path>* by applying a specific lambda expression to every element of the initial *Stream*.

If we have a stream where every element contains its own sequence of elements, and we want to create a stream of these inner elements, we can use the *flatMap()* method:

```
List<Detail> details = new ArrayList<>();
details.add(new Detail());
Stream<String> stream
  = details.stream().flatMap(detail -> detail.getParts().stream());
```

In this example, we have a list of elements of type *Detail*. The *Detail* class contains a field *PARTS*, which is a *List<String>*. With the help of the *flatMap()* method, every element from field *PARTS* will be extracted and added to the new resulting stream. After that, the initial *Stream<Detail>* will be lost.

3.4. Matching

Stream API gives a handy set of instruments to validate elements of a sequence according to some predicate. To do this, one of the following methods can be used: *anyMatch(), allMatch(), noneMatch()*. Their names are self-explanatory. Those are terminal operations that return a *boolean*:

```
boolean isValid = list.stream().anyMatch(element -> element.
contains("h")); // true
boolean isValidOne = list.stream().allMatch(element -> element.
contains("h")); // false
boolean isValidTwo = list.stream().noneMatch(element -> element.
contains("h")); // false
```

For empty streams, the *allMatch()* method with any given predicate will return *true*:

```
Stream.empty().allMatch(Objects::nonNull); // true
```

This is a sensible default, as we can't find any element that doesn't satisfy the predicate. Similarly, the *anyMatch()* method always returns false for empty streams:

```
Stream.empty().anyMatch(Objects::nonNull); // false
```

Again, this is reasonable, as we can't find an element satisfying this condition.

3.5. Reduction

Stream API allows us to reduce a sequence of elements to some value according to a specified function with the help of the *reduce()* method of the type *Stream*. This method takes two parameters: start value first, an accumulator function second.
Imagine that we have a *List<Integer>* and we want to have a sum of all these elements and some initial *Integer* (in this example 23). We can run the following code, and the result will be 26 (23 + 1 + 1 + 1):

```
List<Integer> integers = Arrays.asList(1, 1, 1);
Integer reduced = integers.stream().reduce(23, (a, b) -> a + b);
```

3.6. Collecting

The reduction can also be provided by the *collect()* method of type *Stream*. This operation is very handy for converting a stream to a *Collection* or a *Map*, and representing a stream in the form of a single string. There's a utility class, *Collectors*, which provides a solution for almost all typical collecting operations. For some not trivial tasks, a custom *Collector* can be created:

```
List<String> resultList
  = list.stream().map(element -> element.toUpperCase()).
collect(Collectors.toList());
```

This code uses the terminal *collect()* operation to reduce a *Stream<String>* to the *List<String>*.

4. Conclusion

In this chapter, we briefly touched upon Java streams, definitely one of the most interesting Java 8 features.

There are many more advanced examples of using Streams, but the goal of this chapter is to provide a quick and practical introduction to what we can start doing with the functionality. It's meant to be a starting point for exploring and further learning.

The source code accompanying the chapter is available over on GitHub.

2. Guide to Java 8's Collectors

1. Overview

In this chapter, we'll be going through Java 8's Collectors, which are used at the final step of processing a *Stream*.

2. The *Stream.collect()* Method

Stream.collect() is one of the Java 8's *Stream API's* terminal methods. It allows us to perform mutable fold operations (repackaging elements to some data structures and applying some additional logic, concatenating them, etc.) on data elements held in a *Stream* instance.

The strategy for this operation is provided via the *Collector* interface implementation.

All predefined implementations can be found in the *Collectors* class. It's common practice to use the following static import with them to leverage increased readability:

```
import static java.util.stream.Collectors.*;
```

We can also use single import collectors of our choice:

```
import static java.util.stream.Collectors.toList;
import static java.util.stream.Collectors.toMap;
import static java.util.stream.Collectors.toSet;
```

In the following examples, we'll be reusing the following list:

```
List<String> givenList = Arrays.asList("a", "bb", "ccc", "dd");
```

3.1. *Collectors.toList()*

The *toList* collector can be used for collecting all *Stream* elements into a *List* instance. The important thing to remember is that we can't assume any particular *List* implementation with this method. If we want to have more control over this, we can use *toCollection* instead.

Let's create a *Stream* instance representing a sequence of elements, and then collect them into a *List* instance:

```
List<String> result = givenList.stream()
  .collect(toList());
```

3.1.1. *Collectors.toUnmodifiableList()*

Java 10 introduced a convenient way to accumulate the *Stream* elements into an unmodifiable *List*:

```
List<String> result = givenList.stream()
  .collect(toUnmodifiableList());
```

Now if we try to modify the *result* List, we'll get an *UnsupportedOperationException:*

```
assertThatThrownBy(() -> result.add("foo"))
  .isInstanceOf(UnsupportedOperationException.class);
```

3.2. *Collectors.toSet()*

The *toSet* collector can be used for collecting all *Stream* elements into a *Set* instance. The important thing to remember is that we can't assume any particular *Set* implementation with this method. If we want to have more control over this, we can use *toCollection* instead.

Let's create a *Stream* instance representing a sequence of elements, and then collect them into a *Set* instance:

```
Set<String> result = givenList.stream()
  .collect(toSet());
```

A *Set* doesn't contain duplicate elements. If our collection contains elements equal to each other, they appear in the resulting *Set* only once:

```
List<String> listWithDuplicates = Arrays.asList("a", "bb", "c",
"d", "bb");
Set<String> result = listWithDuplicates.stream().collect(toSet());
assertThat(result).hasSize(4);
```

3.2.1. *Collectors.toUnmodifiableSet()*

Since Java 10, we can easily create an unmodifiable *Set* using the *toUnmodifiableSet()* collector:

```
Set<String> result = givenList.stream()
  .collect(toUnmodifiableSet());
```

Any attempt to modify the result Set will end up with an *UnsupportedOperationException:*

```
assertThatThrownBy(() -> result.add("foo"))
  .isInstanceOf(UnsupportedOperationException.class);
```

3.3. *Collectors.toCollection()*

As we've already noted, when using the *toSet* and *toList* collectors, we can't make any assumptions of their implementations. If we want to use a custom implementation, we'll need to use the *toCollection* collector with a provided collection of our choice.

Let's create a *Stream* instance representing a sequence of elements, and then collect them into a *LinkedList* instance:

```
List<String> result = givenList.stream()
  .collect(toCollection(LinkedList::new))
```

Notice that this won't work with any immutable collections. In such a case, we would need to either write a custom *Collector* implementation, or use *collectingAndThen*.

3.4. *Collectors.toMap()*

The *toMap* collector can be used to collect *Stream* elements into a *Map* instance. To do this, we need to provide two functions:

- keyMapper
- valueMapper

We'll use *keyMapper* to extract a *Map* key from a *Stream* element, and *valueMapper* to extract a value associated with a given key.

Let's collect those elements into a *Map* that stores strings as keys, and their lengths as values:

```
Map<String, Integer> result = givenList.stream()
  .collect(toMap(Function.identity(), String::length))
```

Function.identity() is just a shortcut for defining a function that accepts and returns the same value.

So what happens if our collection contains duplicate elements? Contrary to *toSet*, *toMap* doesn't silently filter duplicates, which is understandable because how would it figure out which value to pick for this key?

```
List<String> listWithDuplicates = Arrays.asList("a", "bb", "c",
"d", "bb");
assertThatThrownBy(() -> {
    listWithDuplicates.stream().collect(toMap(Function.identity(),
String::length));
}).isInstanceOf(IllegalStateException.class);
```

Note that *toMap* doesn't even evaluate whether the values are also equal. If it sees duplicate keys, it immediately throws an *IllegalStateException*.

In such cases with key collision, we should use *toMap* with another signature:

```
Map<String, Integer> result = givenList.stream()
  .collect(toMap(Function.identity(), String::length, (item,
identicalItem) -> item));
```

The third argument here is a *BinaryOperator*, where we can specify how we want collisions to be handled. In this case, we'll just pick any of these two colliding values because we know that the same strings will always have the same lengths too.

3.4.1. *Collectors.toUnmodifiableMap()*

Similar to with Lists and Sets, Java 10 introduced an easy way to collect Stream elements into an unmodifiable *Map*:

```
Map<String, Integer> result = givenList.stream()
  .collect(toUnmodifiableMap(Function.identity(), String::length))
```

As we can see, if we try to put a new entry into a *result Map*, we'll get an *UnsupportedOperationException*:

```
assertThatThrownBy(() -> result.put("foo", 3))
  .isInstanceOf(UnsupportedOperationException.class);
```

3.5. *Collectors.collectingAndThen()*

CollectingAndThen is a special collector that allows us to perform another action on a result straight after collecting ends.

Let's collect *Stream* elements to a List instance, and then convert the result into an *ImmutableList* instance:

```
List<String> result = givenList.stream()
  .collect(collectingAndThen(toList(), ImmutableList::copyOf))
```

3.6. *Collectors.joining()*

Joining collector can be used for joining *Stream<String>* elements.

We can join them together by doing:

```
String result = givenList.stream()
  .collect(joining());
```

This will result in:

```
"abbcccdd"
```

We can also specify custom separators, prefixes, postfixes:

```
String result = givenList.stream()
  .collect(joining(" "));
```

This will result in:

```
"a bb ccc dd"
```

We can also write:

```
String result = givenList.stream()
  .collect(joining(" ", "PRE-", "-POST"));
```

This will result in:

```
"PRE-a bb ccc dd-POST"
```

3.7. *Collectors.counting()*

Counting is a simple collector that allows for the counting of all *Stream* elements.

Now we can write:

```
Long result = givenList.stream()
  .collect(counting());
```

3.8. *Collectors.summarizingDouble/Long/Int()*

SummarizingDouble/Long/Int is a collector that returns a special class containing statistical information about numerical data in a *Stream* of extracted elements.

We can obtain information about string lengths by doing:

```
DoubleSummaryStatistics result = givenList.stream()
  .collect(summarizingDouble(String::length));
```

In this case, the following will be true:

```
assertThat(result.getAverage()).isEqualTo(2);
assertThat(result.getCount()).isEqualTo(4);
assertThat(result.getMax()).isEqualTo(3);
assertThat(result.getMin()).isEqualTo(1);
assertThat(result.getSum()).isEqualTo(8);
```

3.9. *Collectors.averagingDouble/Long/Int()*

AveragingDouble/Long/Int is a collector that simply returns an average of extracted elements.

We can get the average string length by doing:

```
Double result = givenList.stream()
  .collect(averagingDouble(String::length));
```

3.10. *Collectors.summingDouble/Long/Int()*

SummingDouble/Long/Int is a collector that simply returns a sum of extracted elements.

We can get the sum of all string lengths by doing:

```
Double result = givenList.stream()
  .collect(summingDouble(String::length));
```

3.11. *Collectors.maxBy()/minBy()*

MaxBy/MinBy collectors return the biggest/smallest element of a *Stream* according to a provided *Comparator* instance.

We can pick the biggest element by doing:

```
Optional<String> result = givenList.stream()
  .collect(maxBy(Comparator.naturalOrder()));
```

We can see that the returned value is wrapped in an *Optional instance*. This forces users to rethink the empty collection corner case.

3.12. *Collectors.groupingBy()*

GroupingBy collector is used for grouping objects by some property, and then storing the results in a *Map* instance.

We can group them by string length, and store the grouping results in *Set* instances:

```
Map<Integer, Set<String>> result = givenList.stream()
  .collect(groupingBy(String::length, toSet()));
```

This will result in the following being true:

```
assertThat(result)
  .containsEntry(1, newHashSet("a"))
  .containsEntry(2, newHashSet("bb", "dd"))
  .containsEntry(3, newHashSet("ccc"));
```

We can see that the second argument of the *groupingBy* method is a *Collector*. In addition, we're free to use any *Collector* of our choice.

3.13. *Collectors.partitioningBy()*

PartitioningBy is a specialized case of *groupingBy* that accepts a *Predicate* instance, and then collects *Stream* elements into a *Map* instance that stores *Boolean* values as keys, and collections as values. Under the "true" key, we can find a collection of elements matching the given *Predicate*, and under the "false" key, we can find a collection of elements not matching the given *Predicate*.

We can write:

```
Map<Boolean, List<String>> result = givenList.stream()
  .collect(partitioningBy(s -> s.length() > 2))
```

This results in a Map containing:

```
{false=[“a”, “bb”, “dd”], true=[“ccc”]}
```

3.14. *Collectors.teeing()*

Let’s find the maximum and minimum numbers from a given *Stream* using the collectors we’ve learned so far:

```
List<Integer> numbers = Arrays.asList(42, 4, 2, 24);
Optional<Integer> min = numbers.stream().
collect(minBy(Integer::compareTo));
Optional<Integer> max = numbers.stream().
collect(maxBy(Integer::compareTo));
// do something useful with min and max
```

Here we’re using two different collectors, and then combining the results of those two to create something meaningful. Before Java 12, in order to cover such use cases, we had to operate on the given *Stream* twice, store the intermediate results into temporary variables, and then combine those results afterwards.

Fortunately, Java 12 offers a built-in collector that takes care of these steps on our behalf; all we have to do is provide the two collectors and the combiner function.

Since this new collector tees the given stream towards two different directions, it’s called teeing:

```
numbers.stream().collect(teeing(
  minBy(Integer::compareTo), // The first collector
  maxBy(Integer::compareTo), // The second collector
  (min, max) -> // Receives the result from those collectors and
combines them
));
```

This example is available on GitHub in the core-java-12 project.

If we want to write our own Collector implementation, we need to implement the Collector interface, and specify its three generic parameters:

```
1. public interface Collector<T, A, R> {...}
```

T – the type of objects that will be available for collection
A – the type of a mutable accumulator object
R – the type of a final result

Let's write an example Collector for collecting elements into an ImmutableSet instance. We start by specifying the right types:

```
1. private class ImmutableSetCollector<T>
2.   implements Collector<T, ImmutableSet.Builder<T>,
3. ImmutableSet<T>> {...}
```

Since we need a mutable collection for internal collection operation handling, we can't use *ImmutableSet*. Instead, we need to use some other mutable collection, or any other class that could temporarily accumulate objects for us. In this case, we'll go with an *ImmutableSet.Builder*, and now we need to implement 5 methods:

- *Supplier<ImmutableSet.Builder<T>> supplier()*
- *BiConsumer<ImmutableSet.Builder<T>, T> accumulator()*
- *BinaryOperator<ImmutableSet.Builder<T>> combiner()*
- *Function<ImmutableSet.Builder<T>, ImmutableSet<T>> finisher()*
- *Set<Characteristics> characteristics()*

The supplier() method returns a *Supplier* instance that generates an empty accumulator instance. So in this case, we can simply write:

```
1. @Override
2. public Supplier<ImmutableSet.Builder<T>> supplier() {
3.     return ImmutableSet::builder;
4. }
```

The accumulator() method returns a function that's used for adding a new element to an existing *accumulator* object. So let's just use the *Builder's* add method:

```
@Override
public BiConsumer<ImmutableSet.Builder<T>, T> accumulator() {
    return ImmutableSet.Builder::add;
}
```

The combiner() method returns a function that's used for merging two accumulators together:

```
@Override
public BinaryOperator<ImmutableSet.Builder<T>> combiner() {
    return (left, right) -> left.addAll(right.build());
}
```

The finisher() method returns a function that's used for converting an accumulator to final result type. So in this case, we'll just use *Builder's* build method:

```
@Override
public BinaryOperator<ImmutableSet.Builder<T>> combiner() {
    return (left, right) -> left.addAll(right.build());
}
```

The characteristics() method is used to provide Stream with some additional information that will be used for internal optimizations. In this case, we don't pay attention to the elements order in a Set because we'll use *Characteristics.UNORDERED*. To obtain more information regarding this subject, check *Characteristics'* JavaDoc:

```
@Override public Set<Characteristics> characteristics() {
    return Sets.immutableEnumSet(Characteristics.UNORDERED);
}
```

Here is the complete implementation along with the usage:

```
public class ImmutableSetCollector<T>
  implements Collector<T, ImmutableSet.Builder<T>,
ImmutableSet<T>> {

@Override
public Supplier<ImmutableSet.Builder<T>> supplier() {
    return ImmutableSet::builder;
}

@Override
public BiConsumer<ImmutableSet.Builder<T>, T> accumulator() {
    return ImmutableSet.Builder::add;
}

@Override
public BinaryOperator<ImmutableSet.Builder<T>> combiner() {
    return (left, right) -> left.addAll(right.build());
}

@Override
public Function<ImmutableSet.Builder<T>, ImmutableSet<T>>
finisher() {
    return ImmutableSet.Builder::build;
}

@Override
public Set<Characteristics> characteristics() {
    return Sets.immutableEnumSet(Characteristics.UNORDERED);
}

public static <T> ImmutableSetCollector<T> toImmutableSet() {
    return new ImmutableSetCollector<>();
}
```

Finally, here it is in action:

```
List<String> givenList = Arrays.asList("a", "bb", "ccc", "dddd");
ImmutableSet<String> result = givenList.stream()
  .collect(toImmutableSet());
```

5. Conclusion

In this chapter, we explored Java 8's *Collectors* in depth, and demonstrated how to implement one. For more information, you can check out one of these projects that enhances the capabilities of parallel processing in Java.

All code examples are available on GitHub, and more interesting chapters can be read here.

3. Java Stream Filter With Lambda Expression

1. Introduction

In this quick chapter, we'll explore the use of the *Stream.filter()* method when we work with Streams in Java.

We'll look at how to use it, and how to handle special cases with checked exceptions.

The *filter()* method is an intermediate operation of the Stream interface that allows us to filter elements of a stream that match a given Predicate:

Stream<T> filter(Predicate<? super T> predicate)
To see how this works, let's create a Customer class:

```
1.  public class Customer {
2.      private String name;
3.      private int points;
4.      //Constructor and standard getters
5.  }
```

In addition, we'll create a collection of customers:

```
1.  Customer john = new Customer("John P.", 15);
2.  Customer sarah = new Customer("Sarah M.", 200);
3.  Customer charles = new Customer("Charles B.", 150);
4.  Customer mary = new Customer("Mary T.", 1);
5.
6.  List<Customer> customers = Arrays.asList(john, sarah, charles,
7.  mary);
```

2.1. Filtering Collections

A common use case of the *filter()* method is processing collections.
Let's make a list of customers with more than 100 points. To do that, we can use a lambda expression:

```
1.  List<Customer> customersWithMoreThan100Points = customers
2.    .stream()
3.    .filter(c -> c.getPoints() > 100)
4.    .collect(Collectors.toList());
```

We can also use a method reference, which is shorthand for a lambda expression:

```
List<Customer> customersWithMoreThan100Points = customers
  .stream()
  .filter(Customer::hasOverHundredPoints)
  .collect(Collectors.toList());
In this case, we added the hasOverHundredPoints method to our
Customer class:
public boolean hasOverHundredPoints() {
    return this.points > 100;
}
```

In both cases, we get the same result:

```
assertThat(customersWithMoreThan100Points).hasSize(2);
assertThat(customersWithMoreThan100Points).contains(sarah,
charles);
```

2.2. Filtering Collections With Multiple Criteria

Furthermore, we can use multiple conditions with *filter()*. For example, we can filter by points and name:

```
List<Customer> charlesWithMoreThan100Points = customers
  .stream()
  .filter(c -> c.getPoints() > 100 && c.getName().
startsWith("Charles"))
  .collect(Collectors.toList());

assertThat(charlesWithMoreThan100Points).hasSize(1);
assertThat(charlesWithMoreThan100Points).contains(charles);
```

Until now, we've been using the filter with predicates that don't throw an exception. Indeed, the functional interfaces in Java don't declare any checked or unchecked exceptions.

Next, we're going to show some different ways to handle exceptions in lambda expressions.

3.1. Using a Custom Wrapper

First, we'll start by adding a profilePhotoUrl to our Customer:

```
1.  private String profilePhotoUrl;
```

In addition, let's add a simple *hasValidProfilePhoto()* method to check the availability of the profile:

```
1.  public boolean hasValidProfilePhoto() throws IOException {
2.      URL url = new URL(this.profilePhotoUrl);
3.      HttpsURLConnection connection = (HttpsURLConnection) url.
4.  openConnection();
5.      return connection.getResponseCode() == HttpURLConnection.HTTP_
6.  OK;
7.  }
```

We can see that the *hasValidProfilePhoto()* method throws an IOException. Now if we try to filter the customers with this method:

```
1.  List<Customer> customersWithValidProfilePhoto = customers
2.    .stream()
3.    .filter(Customer::hasValidProfilePhoto)
4.    .collect(Collectors.toList());
```

We'll see the following error:

```
1.  Incompatible thrown types java.io.IOException in functional
2.  expression
```

To handle it, one of the alternatives we can use is wrapping it with a try-catch block:

```
List<Customer> customersWithValidProfilePhoto = customers
  .stream()
  .filter(c -> {
      try {
          return c.hasValidProfilePhoto();
      } catch (IOException e) {
          //handle exception
      }
      return false;
  })
  .collect(Collectors.toList());
```

If we need to throw an exception from our predicate, we can wrap it in an unchecked exception like *RuntimeException*.

3.2. Using ThrowingFunction

Alternatively, we can use the ThrowingFunction library.

ThrowingFunction is an open source library that allows us to handle checked exceptions in Java functional interfaces.

Let's start by adding the throwing-function dependency to our pom:

```
<dependency>
    <groupId>com.pivovarit</groupId>
    <artifactId>throwing-function</artifactId>
    <version>1.5.1</version>
</dependency>
```

To handle exceptions in predicates, this library offers us the ThrowingPredicate class, which has the *unchecked()* method to wrap checked exceptions.

Let's see it in action:

```
List customersWithValidProfilePhoto = customers
  .stream()
  .filter(ThrowingPredicate.
unchecked(Customer::hasValidProfilePhoto))
  .collect(Collectors.toList());
```

In this chapter, we illustrated how to use the *filter()* method to process streams. We also explored some alternatives to handle exceptions.

As always, the complete code is available over on GitHub.

4. Working With Maps Using Streams

In this chapter, we'll focus on how to use *Java Streams* to work with Maps. It's worth noting that some of these exercises could be solved using a bidirectional *Map* data structure, but we're interested here in a functional approach.

First, we'll explain the basic idea we'll be using to work with *Maps* and *Streams*. Then we'll present a couple of different problems related to *Maps* and their concrete solutions using *Streams*.

The principal thing to notice is that *Streams* are sequences of elements which can be easily obtained from a *Collection*.

Maps have a different structure, with a mapping from keys to values, without sequence. However, this doesn't mean that we can't convert a *Map* structure into different sequences which then allow us to work in a natural way with the Stream API.

Let's see a few ways of obtaining different *Collections* from a *Map*, which we can then pivot into a *Stream*:

```
1. Map<String, Integer> someMap = new HashMap<>();
```

We can obtain a set of key-value pairs:

```
1. Set<Map.Entry<String, Integer>> entries = someMap.entrySet();
```

We can also get the key set associated with the *Map*:

```
1. Set<Map.Entry<String, Integer>> entries = someMap.entrySet();
```

Or we could work directly with the set of values:

```
1. Collection<Integer> values = someMap.values();
```

These each give us an entry point to process those collections by obtaining streams from them:

```
1. List<Customer> customersWithValidProfilePhoto = customers
2.   .stream()
3.   .filter(Customer::hasValidProfilePhoto)
4.   .collect(Collectors.toList());
```

3.1. Input Data

Let's assume we have a Map:

```
1. Map<String, String> books = new HashMap<>();
2. books.put(
3. "978-0201633610", "Design patterns : elements of reusable object-
4. oriented software");
5. books.put(
6.   "978-1617291999", "Java 8 in Action: Lambdas, Streams, and
7. functional-style programming");
8. books.put("978-0134685991", "Effective Java");
```

We're interested in finding the ISBN for the book titled "Effective Java."

3.2. Retrieving a Match

Since the book title couldn't exist in our Map, we want to be able to indicate that there's no associated ISBN for it. We can use an Optional to express that.

Let's assume for this example that we're interested in any key for a book matching that title:

```
1. Optional<String> optionalIsbn = books.entrySet().stream()
2.   .filter(e -> "Effective Java".equals(e.getValue()))
3.   .map(Map.Entry::getKey)
4.   .findFirst();
5.
6. assertEquals("978-0134685991", optionalIsbn.get());
```

Let's analyze the code. First, **we obtain the *entrySet* from the *Map***, as we saw previously.

We only want to consider the entries with "Effective Java" as the title, so the first intermediate operation will be a filter.

We're not interested in the whole *Map* entry, but in the key of each entry. So the next chained intermediate operation does just that; it's a *map* operation that will generate a new stream as output, which will contain only the keys for the entries that matched the title we were looking for.

As we only want one result, we can apply the *findFirst()* terminal operation, which will provide the initial value in the *Stream* as an *Optional* object.

Let's see an example in which a title doesn't exist:

```
Optional<String> optionalIsbn = books.entrySet().stream()
  .filter(e -> "Non Existent Title".equals(e.getValue()))
  .map(Map.Entry::getKey).findFirst();

assertEquals(false, optionalIsbn.isPresent());
```

3.3. Retrieving Multiple Results

Now let's change the problem to see how we could deal with returning multiple results instead of one.

To have multiple results returned, let's add the following book to our *Map*:

```
books.put("978-0321356680", "Effective Java: Second Edition");
```

So now if we look for *all* books that start with "Effective Java," we'll get more than one result back:

```
List<String> isbnCodes = books.entrySet().stream()
  .filter(e -> e.getValue().startsWith("Effective Java"))
  .map(Map.Entry::getKey)
  .collect(Collectors.toList());

assertTrue(isbnCodes.contains("978-0321356680"));
assertTrue(isbnCodes.contains("978-0134685991"));
```

What we've done in this case is to replace the filter condition to verify if the value in the *Map* starts with "Effective Java" instead of comparing for *String* equality.

This time **we *collect* the results**, instead of just picking the first, and put the matches into a *List*.

Now let's focus on a different problem with maps. **Instead of obtaining *ISBNs* based on the *titles*, we'll try to get *titles* based on the *ISBNs*.**

Let's use the original *Map*. We want to find titles with an ISBN starting with "978-0":

```
1.  List<String> titles = books.entrySet().stream()
2.    .filter(e -> e.getKey().startsWith("978-0"))
3.    .map(Map.Entry::getValue)
4.    .collect(Collectors.toList());
5.
6.  assertEquals(2, titles.size());
7.  assertTrue(titles.contains(
8.    "Design patterns : elements of reusable object-oriented
9.  software"));
10. assertTrue(titles.contains("Effective Java"));
```

This solution is similar to the solutions of our previous set of problems; we stream the entry set, and then filter, map, and collect.

Also like before, if we wanted to return only the first match, after the *map* method, we could call the *findFirst()* method instead of collecting all the results in a *List*.

In this chapter, we demonstrated how to process a *Map* in a functional way.

In particular, we saw that once we switch to using the associated collections to *Maps*, processing using *Streams* becomes much easier and intuitive.

Of course, all of the examples in this chapter can be found in the GitHub project.

5. The Difference Between map() and flatMap()

map() and *flatMap()* APIs stem from functional languages. In Java 8, we can find them in *Optional*, *Stream* and in *CompletableFuture* (although under a slightly different name).

Streams represent a sequence of objects, whereas optionals are classes that represent a value that can be present or absent. We have the *map()* and *flatMap()* methods among other aggregate operations.

Even though both have the same return types, they are quite different. We'll explain these differences by analyzing some examples of streams and optionals.

The *map()* method works well with *Optional* if the function returns the exact type we need:

```
1. Optional<String> s = Optional.of("test");
2. assertEquals(Optional.of("TEST"), s.map(String::toUpperCase));
```

However, in more complex cases, we might be given a function that returns an *Optional* too. In such cases, using *map()* would lead to a nested structure, as the *map()* implementation does an additional wrapping internally.

Let's see another example to get a better understanding of this situation:

```
1. assertEquals(Optional.of(Optional.of("STRING")),
2.   Optional
3.   .of("string")
4.   .map(s -> Optional.of("STRING")));
```

As we can see, we end up with the nested structure *Optional<Optional<String>>*. Although it works, it's pretty cumbersome to use, and doesn't provide any additional null safety, so it's better to keep a flat structure.

That's exactly what *flatMap()* helps us to do:

```
1. assertEquals(Optional.of("STRING"), Optional
2.   .of("string")
3.   .flatMap(s -> Optional.of("STRING")));
```

Both methods work similarly to *Optional*.

The *map()* method wraps the underlying sequence in a *Stream* instance, whereas the *flatMap()* method avoids nested *Stream<Stream<R>>* structure.

Here, *map()* produces a *Stream* consisting of the results of applying the *toUpperCase()* method to the elements of the input *Stream*:

```
List<String> myList = Stream.of("a", "b")
  .map(String::toUpperCase)
  .collect(Collectors.toList());
assertEquals(asList("A", "B"), myList);
```

map() works pretty well in such a simple case. But what if we have something more complex, such as a list of lists as an input?

Let's see how it works:

```
List<List<String>> list = Arrays.asList(
  Arrays.asList("a"),
  Arrays.asList("b"));
System.out.println(list);
```

This snippet prints a list of lists *[[a], [b]]*. Now let's use a *flatMap():*

```
System.out.println(list
  .stream()
  .flatMap(Collection::stream)
  .collect(Collectors.toList()));
```

The result of such a snippet will be flattened to *[a, b]*.

The *flatMap()* method first flattens the input *Stream of Streams* to a *Stream* of *Strings* (for more about flattening, see this chapter). Thereafter, it works similarly to the *map()* method.

Java 8 gives us the opportunity to use the *map()* and *flatMap()* methods that were originally used in functional languages.

We can invoke them on *Streams* and Optionals. These methods help us to get mapped objects by applying the provided mapping function.

As always, the examples in this chapter are available over on GitHub.

6. When to Use a Parallel Stream in Java

Java 8 introduced the Stream API, which makes it easy to iterate over collections as streams of data. It's also very **easy to create streams that execute in parallel, and make use of multiple processor cores**.

We might think that it's always faster to divide the work on more cores, but that's often not the case.

In this chapter, we'll explore the differences between sequential and parallel streams. We'll first look at the default fork-join pool used by parallel streams.

We'll also consider the performance implications of using a parallel stream, including memory locality and splitting/merging costs.

Finally, we'll recommend when it makes sense to convert a sequential stream into a parallel one.

A stream in Java is simply a wrapper around a data source, allowing us to perform bulk operations on the data in a convenient way.

It doesn't store data, or make any changes to the underlying data source. Rather, it adds support for functional-style operations on data pipelines.

2.1. Sequential Streams

By default, **any stream operation in Java is processed sequentially, unless explicitly specified as parallel.**

Sequential streams use a single thread to process the pipeline:

```
List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4);
listOfNumbers.stream().forEach(number ->
    System.out.println(number + " " + Thread.currentThread().
getName())
);
```

The output of this sequential stream is predictable. The list elements will always be printed in an ordered sequence:

```
1 main
2 main
3 main
4 main
```

2.2. Parallel Streams

Any stream in Java can easily be transformed from sequential to parallel.

We can achieve this by **adding the *parallel* method to a sequential stream, or by creating a stream using the *parallelStream* method of a collection:**

```
List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4);
listOfNumbers.parallelStream().forEach(number ->
    System.out.println(number + " " + Thread.currentThread().
getName())
);
```

Parallel streams enable us to execute code in parallel on separate cores. The final result is the combination of each individual outcome.

However, the order of execution is out of our control. It may change every time we run the program:

```
4 ForkJoinPool.commonPool-worker-3
2 ForkJoinPool.commonPool-worker-5
1 ForkJoinPool.commonPool-worker-7
3 main
```

Parallel streams make use of the fork-join framework, and its common pool of worker threads.

The fork-join framework was added to *java.util.concurrent* in Java 7 to handle task management between multiple threads.

3.1. Splitting Source

The fork-join framework is in charge of **splitting the source data between worker threads, and handling callback on task completion.**

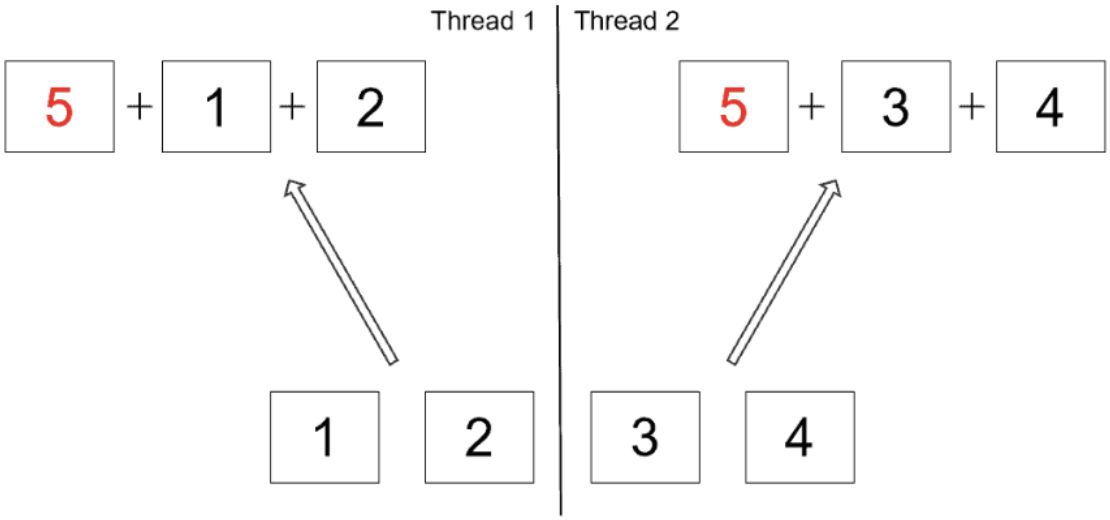
Let's take a look at an example of calculating a sum of integers in parallel.

We'll make use of the reduce method and add five to the starting sum, instead of starting from zero:

```
1. List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4);
2. int sum = listOfNumbers.parallelStream().reduce(5, Integer::sum);
3. assertThat(sum).isNotEqualTo(15);
```

In a sequential stream, the result of this operation would be 15.

But since the *reduce* operation is handled in parallel, the number five actually gets added up in every worker thread:



The actual result might differ depending on the number of threads used in the common fork-join pool.

In order to fix this issue, the number five should be added outside of the parallel stream:

```
1. List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4);
2. int sum = listOfNumbers.parallelStream().reduce(0, Integer::sum)
3. + 5;
4. assertThat(sum).isEqualTo(15);
```

As we can see, we need to be careful about which operations can be run in parallel.

3.2. Common Thread Pool

The number of threads in the common pool is equal to (the number of processor cores -1).

However, the API allows us to specify the number of threads it'll use by passing a JVM parameter:

```
1. -D java.util.concurrent.ForkJoinPool.common.parallelism=4
```

It's important to remember that this is a global setting, and that **it will affect all parallel streams and any other fork-join tasks that use the common pool**. We strongly suggest that this parameter isn't modified unless we have a very good reason for doing so.

3.3. Custom Thread Pool

Besides in the default, common thread pool, it's also possible to run a parallel stream in a custom thread pool:

```
List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4);
ForkJoinPool customThreadPool = new ForkJoinPool(4);
int sum = customThreadPool.submit(
    () -> listOfNumbers.parallelStream().reduce(0, Integer::sum)).
get();
customThreadPool.shutdown();
assertThat(sum).isEqualTo(10);
```

Note that **using the common thread pool is recommended by Oracle**. We should have a very good reason for running parallel streams in custom thread pools.

Parallel processing may be beneficial to fully utilize multiple cores. But we also need to consider the overhead of managing multiple threads, memory locality, splitting the source, and merging the results.

4.1. The Overhead

Let's take a look at an example integer stream.

We'll run a benchmark on a sequential and parallel reduction operation:

```
1. IntStream.rangeClosed(1, 100).reduce(0, Integer::sum);
2. IntStream.rangeClosed(1, 100).parallel().reduce(0, Integer::sum);
```

On this simple sum reduction, converting a sequential stream into a parallel one resulted in worse performance:

```
Benchmark                                                  Mode  Cnt        Score        Error
Units
SplittingCosts.sourceSplittingIntStreamParallel            avgt   25    35476,283 ±    204,446
ns/op
SplittingCosts.sourceSplittingIntStreamSequential          avgt   25       68,274 ±      0,963
ns/op
```

The reason behind this is that sometimes **the overhead of managing threads, sources, and results is a more expensive operation than doing the actual work.**

4.2. Splitting Costs

Splitting the data source evenly is a necessary cost to enable parallel execution, but some data sources split better than others.

Let's demonstrate this using an *ArrayList* and a *LinkedList*:

```
private static final List<Integer> arrayListOfNumbers = new
ArrayList<>();
private static final List<Integer> linkedListOfNumbers = new
LinkedList<>();

static {
    IntStream.rangeClosed(1, 1_000_000).forEach(i -> {
        arrayListOfNumbers.add(i);
        linkedListOfNumbers.add(i);
    });
}
```

We'll run a benchmark on a sequential and parallel reduction operation on the two types of lists:

```
arrayListOfNumbers.stream().reduce(0, Integer::sum)
arrayListOfNumbers.parallelStream().reduce(0, Integer::sum);
linkedListOfNumbers.stream().reduce(0, Integer::sum);
linkedListOfNumbers.parallelStream().reduce(0, Integer::sum);
```

Our results show that converting a sequential stream into a parallel one brings performance benefits only for an *ArrayList*:

```
Benchmark                                                  Mode  Cnt        Score        Error
Units
DifferentSourceSplitting.differentSourceArrayListParallel     avgt   25    2004849,711 ±    5289,437
ns/op
DifferentSourceSplitting.differentSourceArrayListSequential   avgt   25    5437923,224 ±   37398,940
ns/op
DifferentSourceSplitting.differentSourceLinkedListParallel     avgt   25   13561609,611 ±  275658,633
ns/op
DifferentSourceSplitting.differentSourceLinkedListSequential   avgt   25   10664918,132 ±  254251,184
ns/op
```

The reason behind this is that **arrays can split cheaply and evenly**, while *LinkedList* has none of these properties. TreeMap and HashSet split better than *LinkedList*, but not as well as arrays.

4.3. Merging Costs

Every time we split the source for parallel computation, we also need to make sure to combine the results in the end.

Let's run a benchmark on a sequential and parallel stream, with sum and grouping as different merging operations:

```
1. arrayListOfNumbers.stream().reduce(0, Integer::sum);
2. arrayListOfNumbers.stream().parallel().reduce(0, Integer::sum);
3. arrayListOfNumbers.stream().collect(Collectors.toSet());
4. arrayListOfNumbers.stream().parallel().collect(Collectors.
5. toSet())
```

Our results show that converting a sequential stream into a parallel one brings performance benefits only for the sum operation:

```
Benchmark                                                  Mode  Cnt        Score         Error
Units
MergingCosts.mergingCostsGroupingParallel                  avgt   25  135093312,675 ± 4195024,803
ns/op
MergingCosts.mergingCostsGroupingSequential                avgt   25   70631711,489 ± 1517217,320
ns/op
MergingCosts.mergingCostsSumParallel                       avgt   25    2074483,821 ±    7520,402
ns/op
MergingCosts.mergingCostsSumSequential                     avgt   25    5509573,621 ±   60249,942
ns/op
```

The merge operation is really cheap for some operations, such as reduction and addition, but **merge operations, like grouping to sets or maps, can be quite expensive**.

4.4. Memory Locality

Modern computers use a sophisticated multilevel cache to keep frequently used data close to the processor. When a linear memory access pattern is detected, the hardware prefetches the next line of data under the assumption that it will probably be needed soon.

Parallelism brings performance benefits when we can keep the processor cores busy doing useful work. Since waiting for cache misses isn't useful work, we need to consider the memory bandwidth as a limiting factor.

Let's demonstrate this using two arrays, one using a primitive type, and the other using an object data type:

```
1.  private static final int[] intArray = new int[1_000_000];
2.  private static final Integer[] integerArray = new
3.  Integer[1_000_000];
4.
5.  static {
6.      IntStream.rangeClosed(1, 1_000_000).forEach(i -> {
7.          intArray[i-1] = i;
8.          integerArray[i-1] = i;
9.      });
10. }
```

We'll run a benchmark on a sequential and parallel reduction operation on the two arrays:

```
1.  Arrays.stream(intArray).reduce(0, Integer::sum);
2.  Arrays.stream(intArray).parallel().reduce(0, Integer::sum);
3.  Arrays.stream(integerArray).reduce(0, Integer::sum);
4.  Arrays.stream(integerArray).parallel().reduce(0, Integer::sum);
```

Our results show that converting a sequential stream into a parallel one brings slightly more performance benefits when using an array of primitives:

```
Benchmark                                                    Mode  Cnt        Score        Error
Units
MemoryLocalityCosts.localityIntArrayParallel               sequential stream  avgt   25     116247,787
±     283,150  ns/op
MemoryLocalityCosts.localityIntArraySequential               avgt   25     293142,385 ±    2526,892
ns/op
MemoryLocalityCosts.localityIntegerArrayParallel             avgt   25    2153732,607 ±   16956,463
ns/op
MemoryLocalityCosts.localityIntegerArraySequential           avgt   25    5134866,640 ±  148283,942
ns/op
```

An array of primitives brings the best locality possible in Java. In general, **the more pointers we have in our data structure, the more pressure we put on the memory** to fetch the reference objects. This can have a negative effect on parallelization, as multiple cores simultaneously fetch the data from memory.

4.5. The NQ Model

Oracle presented a simple model that can help us determine whether parallelism can offer us a performance boost. In the *NQ* model, *N* stands for the number of source data elements, while Q represents the amount of computation performed per data element.

The larger the product of *N***Q*, the more likely we are to get a performance boost from parallelization. For problems with a trivially small *Q*, such as summing up numbers, the rule of thumb is that *N* should be greater than 10,000. **As the number of computations increases, the data size required to get a performance boost from parallelism decreases.**

4.6. File Search Cost

File Search using parallel streams performs better in comparison to sequential streams. Let's run a benchmark on a sequential and parallel stream for search over 1500 text files:

```
1.  Files.walk(Paths.get("src/main/resources/")).
2.  map(Path::normalize).filter(Files::isRegularFile)
3.        .filter(path -> path.getFileName().toString().endsWith(".
4.  txt")).collect(Collectors.toList());
5.  Files.walk(Paths.get("src/main/resources/")).parallel().
6.  map(Path::normalize).filter(Files::
7.        isRegularFile).filter(path -> path.getFileName().toString().
8.  endsWith(".txt")).
9.        collect(Collectors.toList());
```

Our results show that converting a sequential stream into a parallel one brings slightly more performance benefits when searching a greater number of files:

```
Benchmark                                  Mode  Cnt     Score          Error    Units
FileSearchCost.textFileSearchParallel      avgt   25  10808832.831 ± 446934.773  ns/op
FileSearchCost.textFileSearchSequential    avgt   25  13271799.599 ± 245112.749  ns/op
```

As we've seen, we need to carefully consider when to use parallel streams.

Parallelism can bring performance benefits in certain use cases, but parallel streams can't be considered a magical performance booster. So **sequential streams should still be used as the default during development**.

A sequential stream can be converted to a parallel one when we **have actual performance requirements**. Given those requirements, we should first run a performance measurement, and consider parallelism as a possible optimization strategy.

A large amount of data and many computations done per element indicate that parallelism could be a good option.

On the other hand, a small amount of data, unevenly splitting sources, expensive merge operations, and poor memory locality indicate a potential problem for parallel execution.

In this chapter, we explored the difference between sequential and parallel streams in Java. We learned that parallel streams make use of the default fork-join pool and its worker threads.

Then we saw how parallel streams don't always bring performance benefits. We considered the overhead of managing multiple threads, memory locality, splitting the source, and merging the results. We saw that **arrays are a great data source for parallel execution because they bring the best possible locality and can split cheaply and evenly.**

Finally, we looked at the *NQ* model, and recommended using parallel streams only when we have actual performance requirements.

As always, the source code is available over on GitHub.

7. Guide to Java 8 groupingBy Collector

1. Introduction

In this chapter, **we'll illustrate how the groupingBy collector works using various examples.**

To better understand this chapter, we'll need basic knowledge of Java 8 features. Have a look at the intro to Java 8 Streams and the guide to Java 8's Collectors for these basics.

The Java 8 *Stream* API lets us process collections of data in a declarative way.

The static factory methods *Collectors.groupingBy()* and Collectors.*groupingByConcurrent()* provide us with functionality similar to the *'GROUP BY'* clause in the SQL language. **We can use them for grouping objects by some property and storing results in a *Map* instance**.

The overloaded methods of *groupingBy* are:

- First, with a classification function as the method parameter:

```
static <T,K> Collector<T,?,Map<K,List<T>>>
  groupingBy(Function<? super T,? extends K> classifier)
```

- Second, with a classification function and a second collector as method parameters:

```
static <T,K,A,D> Collector<T,?,Map<K,D>>
  groupingBy(Function<? super T,? K> classifier,
    Collector<? super T,A,D> downstreextends am)
```

- Finally, with a classification function, a supplier method (that provides the *Map* implementation, which contains the end result), and a second collector as method parameters:

```
static <T,K,D,A,M extends Map<K,D>> Collector<T,?,M>
  groupingBy(Function<? super T,? extends K> classifier,
    Supplier<M> mapFactory, Collector<? super T,A,D> downstream)
```

2.1. Example Code Setup

To demonstrate the usage of *groupingBy()*, let's define a *BlogPost* class (we will use a stream of *BlogPost* objects):

```
class BlogPost {
    String title;
    String author;
    BlogPostType type;
    int likes;
}
```

Next, the *BlogPostType*:

```
enum BlogPostType {
    NEWS,
    REVIEW,
    GUIDE
}
```

Then the *List* of *BlogPost* objects:

```
List<BlogPost> posts = Arrays.asList( ... );
```

Let's also define a *Tuple* class to group posts by the combination of their *type* and *author* attributes:

```
class Tuple {
    BlogPostType type;
    String author;
}
```

2.2. Simple Grouping by a Single Column

Let's start with the simplest *groupingBy* method, which only takes a classification function as its parameter. A classification function is applied to each element of the stream.

We use the value returned by the function as a key to the map that we get from the *groupingBy* collector.

To group the blog posts in the blog post list by their *type*:

```
1. Map<BlogPostType, List<BlogPost>> postsPerType = posts.stream()
2.   .collect(groupingBy(BlogPost::getType));
```

2.3. *groupingBy* With a Complex *Map* Key Type

The classification function isn't limited to returning only a scalar or String value. The key of the resulting map could be any object, as long as we make sure that we implement the necessary *equals* and *hashcode* methods.

To **group using two fields as keys, we can use the *Pair* class provided in the *javafx.util* or *org.apache.commons.lang3.tuple* packages.**

For example, to group the blog posts in the list by the type and author combined in an Apache Commons *Pair* instance:

```
1. Map<Pair<BlogPostType, String>, List<BlogPost>>
2. postsPerTypeAndAuthor = posts.stream()
3.   .collect(groupingBy(post -> new ImmutablePair<>(post.getType(),
4. post.getAuthor())));
```

Similarly, we can use the Tuple class defined before; this class can be easily generalized to include more fields as needed. The previous example using a Tuple instance will be:

```
Map<Pair<BlogPostType, String>, List<BlogPost>>
postsPerTypeAndAuthor = posts.stream()
  .collect(groupingBy(post -> new ImmutablePair<>(post.getType(),
post.getAuthor())));
```

Java 16 has introduced the concept of a record as a new form of generating immutable Java classes.

The *record* feature provides us with a simpler, clearer, and safer way to do *groupingBy* than the Tuple. For example, we've defined a *record* instance in the *BlogPost*:

```
public class BlogPost {
    private String title;
    private String author;
    private BlogPostType type;
    private int likes;
    record AuthPostTypesLikes(String author, BlogPostType type,
int likes) {};

    // constructor, getters/setters
}
```

Now it's very simple to group the *BlogPost* in the list by the type, author, and likes using the *record* instance:

```
Map<BlogPost.AuthPostTypesLikes, List<BlogPost>>
postsPerTypeAndAuthor = posts.stream()
  .collect(groupingBy(post -> new BlogPost.
AuthPostTypesLikes(post.getAuthor(), post.getType(), post.
getLikes())));
```

2.4. Modifying the Returned *Map* Value Type

The second overload of *groupingBy* takes an additional second collector (downstream collector) that's applied to the results of the first collector.

When we specify a classification function, but not a downstream collector, the *toList()* collector is used behind the scenes.

Let's use the *toSet()* collector as the downstream collector, and get a *Set* of blog posts (instead of a *List*):

```
Map<BlogPostType, Set<BlogPost>> postsPerType = posts.stream()
  .collect(groupingBy(BlogPost::getType, toSet()));
```

2.5. Grouping by Multiple Fields

A different application of the downstream collector is to do a secondary *groupingBy* to the results of the first group by.

To group the *List* of *BlogPosts* first by *author* and then by *type*:

```
Map<String, Map<BlogPostType, List>> map = posts.stream()
  .collect(groupingBy(BlogPost::getAuthor,
groupingBy(BlogPost::getType)));
```

2.6. Getting the Average From Grouped Results

By using the downstream collector, we can apply aggregation functions in the results of the classification function.

For instance, to find the average number of *likes* for each blog post *type*:

```
Map<BlogPostType, Double> averageLikesPerType = posts.stream()
  .collect(groupingBy(BlogPost::getType,
averagingInt(BlogPost::getLikes)));
```

2.7. Getting the Sum From Grouped Results

To calculate the total sum of likes for each type:

```
Map<BlogPostType, Integer> likesPerType = posts.stream()
  .collect(groupingBy(BlogPost::getType,
summingInt(BlogPost::getLikes)));
```

2.8. Getting the Maximum or Minimum From Grouped Results

Another aggregation that we can perform is to get the blog post with the maximum number of likes:

```
Map<BlogPostType, Optional<BlogPost>> maxLikesPerPostType =
posts.stream()
  .collect(groupingBy(BlogPost::getType,
  maxBy(comparingInt(BlogPost::getLikes))));
```

Similarly, we can apply the *minBy* downstream collector to obtain the blog post with the minimum number of *likes*.

Note that the *maxBy* and *minBy* collectors take into account the possibility that the collection to which they're applied could be empty. This is why the value type in the map is *Optional<BlogPost>*.

2.9. Getting a Summary for an Attribute of Grouped Results

The *Collectors* API offers a summarizing collector that we can use in cases when we need to calculate the count, sum, minimum, maximum and average of a numerical attribute at the same time.

Let's calculate a summary for the likes attribute of the blog posts for each different type:

```
Map<BlogPostType, IntSummaryStatistics> likeStatisticsPerType =
posts.stream()
  .collect(groupingBy(BlogPost::getType,
  summarizingInt(BlogPost::getLikes)));
```

The *IntSummaryStatistics* object for each type contains the count, sum, average, min, and max values for the *likes* attribute. Additional summary objects exist for double and long values.

2.10. Aggregating Multiple Attributes of a Grouped Result

In the previous sections we saw how to aggregate one field at a time. **There are some techniques that we can follow to do aggregations over multiple fields**.

The first approach is to use *Collectors::collectingAndThen* for the downstream collector of *groupingBy*. For the first parameter of *collectingAndThen*, we collect the stream into a list using *Collectors::toList*. The second parameter applies the finishing transformation; we can use it with any of the *Collectors*' class methods that support aggregations to get our desired results.

For example, let's group by *author*, and for each one we count the number of *titles*, list the *titles*, and provide a summary statistics of the *likes*. To accomplish this, we start by adding a new record to the *BlogPost*:

```
1.  public class BlogPost {
2.      // ...
3.      record PostCountTitlesLikesStats(long postCount, String
4.  titles, IntSummaryStatistics likesStats){};
5.       // ...
6.  }
```

The implementation of *groupingBy* and *collectingAndThen* will be:

```
1.  Map<String, BlogPost.PostCountTitlesLikesStats> postsPerAuthor =
2.  posts.stream()
3.    .collect(groupingBy(BlogPost::getAuthor,
4.  collectingAndThen(toList(), list -> {
5.      long count = list.stream()
6.        .map(BlogPost::getTitle)
7.        .collect(counting());
8.      String titles = list.stream()
9.        .map(BlogPost::getTitle)
10.       .collect(joining(" : "));
11.     IntSummaryStatistics summary = list.stream()
12.       .collect(summarizingInt(BlogPost::getLikes));
13.     return new BlogPost.PostCountTitlesLikesStats(count, titles,
14. summary);
15.   })));
```

In the first parameter of *collectingAndThen* we get a list of *BlogPost*. We'll use it in the finishing transformation as an input to the lambda function to calculate the values to generate *PostCountTitlesLikesStats*.

To get the information for a given *author* is as simple as:

```
1. BlogPost.PostCountTitlesLikesStats result = postsPerAuthor.
2. get("Author 1");
3. assertThat(result.postCount()).isEqualTo(3L);
4. assertThat(result.titles()).isEqualTo("News item 1 : Programming
5. guide : Tech review 2");
6. assertThat(result.likesStats().getMax()).isEqualTo(20);
7. assertThat(result.likesStats().getMin()).isEqualTo(15);
8. assertThat(result.likesStats().getAverage()).isEqualTo(16.666d,
9. offset(0.001d));
```

We can also do more sophisticated aggregations if we use *Collectors::toMap* to collect and aggregate the elements of the stream.

Let's consider a simple example where we want to group the *BlogPost* elements by *author*, and concatenate the *titles* with an upper bounded sum of *like* scores.

First, we'll create the record that's going to encapsulate our aggregated result:

```
1. public class BlogPost {
2.     // ...
3.     record TitlesBoundedSumOfLikes(String titles, int
4. boundedSumOfLikes) {};
5.     // ...
6. }
```

Then we group and accumulate the stream in the following manner:

```
int maxValLikes = 17;
Map<String, BlogPost.TitlesBoundedSumOfLikes> postsPerAuthor =
posts.stream()
  .collect(toMap(BlogPost::getAuthor, post -> {
    int likes = (post.getLikes() > maxValLikes) ? maxValLikes :
post.getLikes();
    return new BlogPost.TitlesBoundedSumOfLikes(post.getTitle(),
likes);
  }, (u1, u2) -> {
    int likes = (u2.boundedSumOfLikes() > maxValLikes) ?
maxValLikes : u2.boundedSumOfLikes();
    return new BlogPost.TitlesBoundedSumOfLikes(u1.
titles().toUpperCase() + " : " + u2.titles().toUpperCase(),
u1.boundedSumOfLikes() + likes);
  }));
```

The first parameter of *toMap* groups the keys applying *BlogPost::getAuthor.*

The second parameter transforms the values of the map using the lambda function to convert each *BlogPost* into a *TitlesBoundedSumOfLikes* record.

The third parameter of *toMap* deals with duplicate elements for a given key, and here we use another lambda function to concatenate the *titles* and sum the *likes* with a max allowed value specified in *maxValLikes*.

2.11. Mapping Grouped Results to a Different Type

We can achieve more complex aggregations by applying a *mapping* downstream collector to the results of the classification function.

Let's get a concatenation of the *titles* of the posts for each blog post *type*:

```
Map<BlogPostType, String> postsPerType = posts.stream()
  .collect(groupingBy(BlogPost::getType,
  mapping(BlogPost::getTitle, joining(", ", "Post titles: [",
"]"))));
```

What we've done here is to map each *BlogPost* instance to its *title*, and then reduce the stream of post titles to a concatenated *String*. In this example, the type of the *Map* value is also different from the default *List* type.

2.12. Modifying the Return *Map* Type

When using the *groupingBy* collector, we can't make assumptions about the type of the returned *Map*. If we want to be specific about which type of *Map* we want to get from the group by, then we can use the third variation of the *groupingBy* method that allows us to change the type of the *Map* by passing a *Map* supplier function.

Let's retrieve an *EnumMap* by passing an *EnumMap* supplier function to the *groupingBy* method:

```
groupingBy method:
EnumMap<BlogPostType, List<BlogPost>> postsPerType = posts.
stream()
  .collect(groupingBy(BlogPost::getType,
  () -> new EnumMap<>(BlogPostType.class), toList()));
```

Similar to *groupingBy* is the *groupingByConcurrent* collector, which leverages multi-core architectures. This collector has three overloaded methods that take exactly the same arguments as the respective overloaded methods of the *groupingBy* collector. The return type of the *groupingByConcurrent* collector, however, must be an instance of the *ConcurrentHashMap* class or a subclass of it.

To do a grouping operation concurrently, the stream needs to be parallel:

```
1.  ConcurrentMap<BlogPostType, List<BlogPost>> postsPerType = posts.
2.  parallelStream()
3.    .collect(groupingByConcurrent(BlogPost::getType));
```

If we choose to pass a *Map* supplier function to the *groupingByConcurrent* collector, then we need to make sure that the function returns either a *ConcurrentHashMap* or a subclass of it.

Java 9 introduced two new collectors that work well with *groupingBy*; more information about them can be found here.

In this chapter, we explored the usage of the *groupingBy* collector offered by the Java 8 *Collectors* API.

We learned how *groupingBy* can be used to classify a stream of elements based on one of their attributes, and how the results of this classification can be further collected, mutated, and reduced to final containers.

The complete implementation of the examples in this chapter can be found in the GitHub project.

8. Guide to Stream.reduce()

The Stream API provides a rich repertoire of intermediate, reduction, and terminal functions that also support parallelization.

More specifically, **reduction stream operations allow us to produce one single result from a sequence of elements** by repeatedly applying a combining operation to the elements in the sequence.

In this chapter, **we'll look at the general-purpose** Stream.reduce() **operation** and demonstrate it with some concrete use cases.

Before we look deeper into using the *Stream.reduce()* operation, let's break down the operation's participant elements into separate blocks. That way, we'll understand more easily the role that each one plays:

- *Identity* – an element that's the initial value of the reduction operation and the default result if the stream is empty
- *Accumulator* – a function that takes two parameters: a partial result of the reduction operation and the next element of the stream
- *Combiner* – a function used to combine the partial result of the reduction operation when the reduction is parallelized, or when there's a mismatch between the types of the accumulator arguments and the types of the accumulator implementation

To better understand the functionality of the identity, accumulator, and combiner elements, let's look at some basic examples:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
int result = numbers
  .stream()
  .reduce(0, (subtotal, element) -> subtotal + element);
assertThat(result).isEqualTo(21);
```

In this case, **the *Integer* value 0 is the identity**. It stores the initial value of the reduction operation, and also the default result when the stream of *Integer* values is empty.

Likewise, **the lambda is the accumulator**, since it takes the partial sum of *Integer* values and the next element in the stream:

```
subtotal, element -> subtotal + element
```

To make the code even more concise, we can use a method reference instead of a lambda expression:

```
int result = numbers.stream().reduce(0, Integer::sum);
assertThat(result).isEqualTo(21);
```

Of course, we can use a *reduce()* operation on streams holding other types of elements.

For instance, we can use *reduce()* on an array of *String* elements, and join them into a single result:

```
List<String> letters = Arrays.asList("a", "b", "c", "d", "e");
String result = letters
  .stream()
  .reduce("", (partialString, element) -> partialString +
element);
assertThat(result).isEqualTo("abcde");
```

Similarly, we can switch to the version that uses a method reference:

```
String result = letters.stream().reduce("", String::concat);
assertThat(result).isEqualTo("abcde");
```

Let's use the *reduce()* operation for joining the uppercase elements of the *letters* array:

```
String result = letters
  .stream()
  .reduce(
    "", (partialString, element) -> partialString.toUpperCase() +
element.toUpperCase());
assertThat(result).isEqualTo("ABCDE");
```

In addition, we can use *reduce()* in a parallelized stream (more on this later):

```
List<Integer> ages = Arrays.asList(25, 30, 45, 28, 32);
int computedAges = ages.parallelStream().reduce(0, (a, b) -> a +
b, Integer::sum);
```

When a stream executes in parallel, the Java runtime splits the stream into multiple substreams. In such cases, **we need to use a function to combine the results of the substreams into a single one. This is the role of the combiner**. In the above snippet, it's the *Integer::sum* method reference.

Funnily enough, this code won't compile:

```
List<User> users = Arrays.asList(new User("John", 30), new
User("Julie", 35));
int computedAges =
  users.stream().reduce(0, (partialAgeResult, user) ->
partialAgeResult + user.getAge());
```

In this case, we have a stream of *User* objects, and the types of the accumulator arguments are *Integer* and *User*. However, the accumulator implementation is a sum of *Integers*, so the compiler just can't infer the type of the *user* parameter.

We can fix this issue by using a combiner:

```
1. int result = users.stream()
2.   .reduce(0, (partialAgeResult, user) -> partialAgeResult + user.
3. getAge(), Integer::sum);
4. assertThat(result).isEqualTo(65);
```

To put it simply, if we use sequential streams, and the types of the accumulator arguments and the types of its implementation match, we don't need to use a combiner.

As we learned before, we can use *reduce()* on parallelized streams.

When we use parallelized streams, we should make sure that *reduce()* or any other aggregate operations executed on the streams are:

- associative: the result isn't affected by the order of the operands
- non-interfering: the operation doesn't affect the data source
- stateless and deterministic: the operation doesn't have state, and produces the same output for a given input

We should fulfill all these conditions to prevent unpredictable results.

As expected, operations performed on parallelized streams, including *reduce()*, are executed in parallel, thus taking advantage of multi-core hardware architectures.

For obvious reasons, **parallelized streams are much more performant than the sequential counterparts.** Even so, they can be overkill if the operations applied to the stream aren't expensive, or the number of elements in the stream is small.

Of course, parallelized streams are the right way to go when we need to work with large streams and perform expensive aggregate operations.

Let's create a simple JMH (the Java Microbenchmark Harness) benchmark test, and compare the respective execution times when using the *reduce()* operation on a sequential and parallelized stream:

```
@State(Scope.Thread)
private final List<User> userList = createUsers();

@Benchmark
public Integer executeReduceOnParallelizedStream() {
    return this.userList
      .parallelStream()
      .reduce(
        0, (partialAgeResult, user) -> partialAgeResult + user.
getAge(), Integer::sum);
}

@Benchmark
public Integer executeReduceOnSequentialStream() {
    return this.userList
      .stream()
      .reduce(
        0, (partialAgeResult, user) -> partialAgeResult + user.
getAge(), Integer::sum);
}
```

In the above JMH benchmark, we compare execution average times. We simply create a *List* containing a large number of *User* objects. Next, we call *reduce()* on a sequential and parallelized stream, and check that the latter performs faster than the former (in seconds per operation).

These are our benchmark results:

```
Benchmark                                                  Mode  Cnt  Score    Error  Units
JMHStreamReduceBenchMark.executeReduceOnParallelizedStream  avgt    5  0,007 ±  0,001   s/op
JMHStreamReduceBenchMark.executeReduceOnSequentialStream    avgt    5  0,010 ±  0,001   s/op
```

In the above examples, the *reduce()* operation doesn't throw any exceptions, but it could.

For instance, say we need to divide all the elements of a stream by a supplied factor and then sum them:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
int divider = 2;
int result = numbers.stream().reduce(0, a / divider + b / divider);
```

This will work, as long as the *divider* variable isn't zero. If it is zero, *reduce()* will throw an *ArithmeticException* exception: divide by zero.

We can easily catch the exception and do something useful with it, such as logging it, recovering from it, etc. depending on the use case, by using a try/catch block:

```
public static int divideListElements(List<Integer> values, int
divider) {
    return values.stream()
      .reduce(0, (a, b) -> {
          try {
              return a / divider + b / divider;
          } catch (ArithmeticException e) {
            LOGGER.log(Level.INFO, "Arithmetic Exception: Division
by Zero");
          }
          return 0;
      });
}
```

While this approach will work, **we polluted the lambda expression with the *try/catch* block.** We no longer have the clean one-liner that we had before.

To fix this issue, we can use the extract function refactoring technique **and extract the *try/catch* block into a separate method:**

```
1.  private static int divide(int value, int factor) {
2.      int result = 0;
3.      try {
4.          result = value / factor;
5.      } catch (ArithmeticException e) {
6.          LOGGER.log(Level.INFO, "Arithmetic Exception: Division by
7.  Zero");
8.      }
9.      return result
10. }
```

Now the implementation of the *divideListElements()* method is again clean and streamlined:

```
1.  public static int divideListElements(List<Integer> values, int
2.  divider) {
3.      return values.stream().reduce(0, (a, b) -> divide(a, divider)
4.  + divide(b, divider));
5.  }
```

Assuming that *divideListElements()* is a utility method implemented by an abstract *NumberUtils* class, we can create a unit test to check the behavior of the *divideListElements()* method:

```
1.  List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
2.  assertThat(NumberUtils.divideListElements(numbers, 1)).
3.  isEqualTo(21);
```

Let's also test the *divideListElements()* method when the supplied *List* of *Integer* values contains a 0:

```
1.  List<Integer> numbers = Arrays.asList(0, 1, 2, 3, 4, 5, 6);
2.  assertThat(NumberUtils.divideListElements(numbers, 1)).
3.  isEqualTo(21);
```

Finally, let's test the method implementation when the divider is 0 too:

```
1.  List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
2.  assertThat(NumberUtils.divideListElements(numbers, 0)).
3.  isEqualTo(0);
```

We **can also use *Stream.reduce()* with custom objects that contain non-primitive fields.** To do so, we need to provide a relevant *identity*, *accumulator*, and *combiner* for the data type.

Suppose our *User* is part of a review website. Each of our *Users* can possess one *Rating*, which is averaged over many *Reviews*.

First, let's start with our *Review* object.

Each *Review* should contain a simple comment and score:

```
1.  public class Review {
2.
3.      private int points;
4.      private String review;
5.
6.      // constructor, getters and setters
7.  }
```

Next, we need to define our *Rating*, which will hold our reviews alongside a *points* field. As we add more reviews, this field will increase or decrease accordingly:

```
1.  public class Rating {
2.
3.      double points;
4.      List<Review> reviews = new ArrayList<>();
5.
6.      public void add(Review review) {
7.          reviews.add(review);
8.          computeRating();
9.      }
10.
11.     private double computeRating() {
12.         double totalPoints =
13.                 reviews.stream().map(Review::getPoints).reduce(0,
14. Integer::sum);
15.         this.points = totalPoints / reviews.size();
16.         return this.points;
17.     }
18.
19.     public static Rating average(Rating r1, Rating r2) {
20.         Rating combined = new Rating();
21.         combined.reviews = new ArrayList<>(r1.reviews);
21.         combined.reviews.addAll(r2.reviews);
22.         combined.computeRating();
23.         return combined;
24.     }
```

We've also added an *average* function to compute an average based on the two input *Ratings*. This will work nicely for our *combiner* and *accumulator* components.

Next, let's define a list of *Users*, each with their own sets of reviews:

```
User john = new User("John", 30);
john.getRating().add(new Review(5, ""));
john.getRating().add(new Review(3, "not bad"));
User julie = new User("Julie", 35);
john.getRating().add(new Review(4, "great!"));
john.getRating().add(new Review(2, "terrible experience"));
john.getRating().add(new Review(4, ""));
List<User> users = Arrays.asList(john, julie);
```

Now that John and Julie are accounted for, let's use *Stream.reduce()* to compute an average rating across both users.

As an *identity*, let's return a new *Rating* if our input list is empty:

```
Rating averageRating = users.stream()
  .reduce(new Rating(),
    (rating, user) -> Rating.average(rating, user.getRating()),
    Rating::average);
```

If we do the math, we should find that the average score is 3.6:

```
assertThat(averageRating.getPoints()).isEqualTo(3.6);
```

In this chapter, **we learned how to use the *Stream.reduce()* operation**.

In addition, we learned how to perform reductions on sequential and parallelized streams, as well as how to handle exceptions while reducing.

As usual, all the code samples shown in this chapter are available over on GitHub.