Java Streams Example

ANJU MUNOTH

Java streams

- Major new feature in Java 8
- java.util.stream --Contains classes for processing sequences of elements.
- 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.
- Doesn't store data, or make any changes to the underlying data source.
- Adds support for functional-style operations on data pipelines.

Stream Creation

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

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

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

```
Stream stream = list.stream()
```

- Once created, the instance **will not modify its source,** therefore allowing the creation of multiple instances from a single source.

Multi-threading With Stream

- 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:

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

Empty Stream

- Use the empty() method for creation of an empty stream:

```
Stream<String> streamEmpty = Stream.empty();
```

- Use the empty() method upon creation to avoid returning null for streams with no element

```
public Stream<String> streamOf(List<String> list)
{
    return list == null || list.isEmpty() ? Stream.empty() : list.stream();
}
```

Stream of Collection

- Can also create a stream of any type of Collection (Collection, List, Set):

```
Collection<String> collection = Arrays.asList("a", "b", "c");
Stream<String> streamOfCollection = collection.stream();
```

Stream of Array

- An array can also be the source of a stream:

```
Stream<String> streamOfArray = Stream.of("a", "b", "c");
```

- Can also create a stream out of an existing array or of part of an array:

```
String[] arr = new String[]{"a", "b", "c"};
Stream<String> streamOfArrayFull = Arrays.stream(arr);
Stream<String> streamOfArrayPart = Arrays.stream(arr, 1, 3);
```

Stream.builder()

- When builder is used, the desired type should be additionally specified in the right part of the statement, otherwise the build() method will create an instance of the Stream<Object>:

```
Stream<String> streamBuilder =  Stream.<String>builder().add("a").add("b").add("c").build();
```

Stream.generate()

- generate() method accepts a Supplier<T> for element generation.
- As the resulting stream is infinite, the developer should specify the desired size, or the generate() method will work until it reaches the memory limit:

```
Stream<String> streamGenerated =  Stream.generate(() -> "element").limit(10);
```

- code above creates a sequence of ten strings with the value *"element."*

Stream.iterate()

- Another way of creating an infinite stream is by using the iterate() method:

```
Stream<Integer> streamIterated = Stream.iterate(40, n -> n + 2).limit(20);
```

- First element of the resulting stream is the first parameter of the iterate() method.
- When creating every following element, the specified function is applied to the previous element.
- In the example above the second element will be 42.

Stream of Primitives

- Java 8 offers the possibility to create streams out of three primitive types: int, long and double.
- As Stream<T> is a generic interface, and there is no way to use primitives as a type parameter with generics, three new special interfaces were created: **IntStream, LongStream, DoubleStream.**
- Using the new interfaces alleviates unnecessary auto-boxing, which allows for increased productivity

Stream of Primitives

```
IntStream intStream = IntStream.range(1, 3);
LongStream longStream = LongStream.rangeClosed(1, 3);
```

- ***range(int startInclusive, int endExclusive)*** method creates an ordered stream from the first parameter to the second parameter.
  - Increments the value of subsequent elements with the step equal to 1.
  - Result doesn't include the last parameter, it is just an upper bound of the sequence.
- The ***rangeClosed(int startInclusive, int endInclusive)*** method does the same thing with only one difference, the second element is included.
- Can use these two methods to generate any of the three types of streams of primitives.

Stream of Primitives

- Since Java 8, the *Random* class provides a wide range of methods for generating streams of primitives.
- For example, the following code creates a *DoubleStream*, which has three elements:

```
Random random = new Random();
DoubleStream doubleStream = random.doubles(3);
```

Stream of String

- Can also use String as a source for creating a stream with the help of the chars() method of the String class.
- Since there is no interface for CharStream in JDK, we use the IntStream to represent a stream of chars instead.

```
IntStream streamOfChars = "abc".chars();
```

- The following example breaks a String into sub-strings according to specified RegEx:

```
Stream<String> streamOfString =  Pattern.compile(", ").splitAsStream("a, b, c");
```

Stream of File

- Java NIO class Files allows us to generate a Stream<String> of a text file through the lines() method.
- Every line of the text becomes an element of the stream:

```
Path path = Paths.get("C:\\file.txt");
Stream<String> streamOfStrings = Files.lines(path);
Stream<String> streamWithCharset =  Files.lines(path, Charset.forName("UTF-8"));
```

- The Charset can be specified as an argument of the lines() method.

Referencing a Stream

- Can instantiate a stream, and have an accessible reference to it, as long as only intermediate operations are called.
- Executing a terminal operation makes a stream inaccessible.

Referencing a Stream

- Besides its unnecessary verbosity, technically the following code is valid:

```
Stream<String> stream =   Stream.of("a", "b", "c").filter(element -> element.contains("b"));
Optional<String> anyElement = stream.findAny();
```

- However, an attempt to reuse the same reference after calling the terminal operation will trigger the IllegalStateException:

```
Optional<String> firstElement = stream.findFirst();
```

As the IllegalStateException is a RuntimeException, a compiler will not signalize about a problem. So it is very important to remember that Java 8 streams can't be reused.

Referencing a Stream

- Designed streams to apply a finite sequence of operations to the source of elements in a functional style, not to store elements.
- So to make the previous code work properly, some changes should be made:

```
List<String> elements =
  Stream.of("a", "b", "c").filter(element -> element.contains("b")).collect(Collectors.toList());
Optional<String> anyElement = elements.stream().findAny();
Optional<String> firstElement = elements.stream().findFirst();
```

Stream Pipeline

- To perform a sequence of operations over the elements of the data source and aggregate their results, we need three parts: **the source, intermediate operation(s) and a terminal operation.**
- Intermediate operations return a new modified stream.
- For example, to create a new stream of the existing one without few elements, the skip() method should be used:

```
Stream<String> onceModifiedStream =  Stream.of("abcd", "bbcd", "cbcd").skip(1);
```

- If we need more than one modification, we can chain intermediate operations.
- Let's assume that we also need to substitute every element of the current Stream<String> with a sub-string of the first few chars.
- Can do this by chaining the skip() and map() methods:

```
Stream<String> twiceModifiedStream =  stream.skip(1).map(element -> element.substring(0, 3));
```

Stream Pipeline

- A stream by itself is worthless;
- the user is interested in the result of the terminal operation, which can be a value of some type or an action applied to every element of the stream.
- Can only use **one terminal operation per stream.**

- The correct and most convenient way to use streams is by a stream pipeline, which is a **chain of the stream source, intermediate operations, and a terminal operation:**

```
List<String> list = Arrays.asList("abc1", "abc2", "abc3");
long size = list.stream().skip(1)
  .map(element -> element.substring(0, 3)).sorted().count();
```

Lazy Invocation

- Intermediate operations are lazy. This means that they will be invoked only if it is necessary for the terminal operation execution.

Lazy Invocation

```
private long counter;

private void wasCalled() {
    counter++;
}
```

```
List<String> list = Arrays.asList(“abc1”, “abc2”, “abc3”);
counter = 0;
Stream<String> stream = list.stream().filter(element -> {
    wasCalled();
    return element.contains("2");
});
```

As we have a source of three elements, we can assume that the *filter()* method will be called three times, and the value of the *counter* variable will be 3. However, running this code **doesn’t change *counter* at all, it is still zero,** so the *filter()* method wasn’t even called once. The reason why is missing of the terminal operation.

Lazy Invocation

```
Optional<String> stream = list.stream().filter(element -> {
    log.info("filter() was called");
    return element.contains("2");
}).map(element -> {
    log.info("map() was called");
    return element.toUpperCase();
}).findFirst();
```

Resulting log shows that we called the *filter()* method twice and the *map()* method once. This is because the pipeline executes vertically. In our example, the first element of the stream didn't satisfy the filter's predicate. Then we invoked the *filter()* method for the second element, which passed the filter. Without calling the *filter()* for the third element, we went down through the pipeline to the *map()* method.

The *findFirst()* operation satisfies by just one element. So, in this particular example, the lazy invocation allowed us to avoid one method call for *filter()*.

Order of Execution

- From the performance point of view, the right order is one of the most important aspects of chaining operations in the stream pipeline:

```
long size = list.stream().map(element -> {
    wasCalled();
    return element.substring(0, 3);
}).skip(2).count();
```

- Execution of this code will increase the value of the counter by three.
- This means that we called the map() method of the stream three times, but the value of the size is one.
- So the resulting stream has just one element, and we executed the expensive map() operations for no reason two out of the three times.

Order of Execution

- If we change the order of the skip() and the map() methods, the counter will increase by only one. So we will call the map() method only once:

```
long size = list.stream().skip(2).map(element -> {
    wasCalled();
    return element.substring(0, 3);
}).count();
```

- Rule: **intermediate operations which reduce the size of the stream should be placed before operations which are applying to each element.**
- Need to keep methods such as skip(), filter(), and distinct() at the top of our stream pipeline.

Iterating

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

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

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

```
long count = list.stream().distinct().count();
boolean isExist = list.stream().anyMatch(element -> element.contains("a"))
```

Filtering

- filter() method allows us to pick a stream of elements that satisfy a predicate

```
ArrayList<String> list = new ArrayList<>();
list.add("One");
list.add("OneAndOnly");
list.add("Derek");
list.add("Change");
list.add("factory");
list.add("justBefore");
list.add("Italy");
list.add("Italy");
list.add("Thursday");
list.add("");
list.add("");
```

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

Mapping

- To convert elements of a Stream by applying a special function to them, and to collect these new elements into a Stream,

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

flatMap()

- If we have a stream where every element contains its own sequence of elements, and want to create a stream of these inner elements, -- use the flatMap() method
- Have a list of elements of type Detail.
- Detail class contains a field PARTS, which is a List.
- 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 will be lost

```
List details = new ArrayList<>(); details.add(new Detail());

Stream stream = details.stream().flatMap(detail -> detail.getParts().stream());
```

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().
- **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

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

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

Stream Reduction

- The API has many terminal operations which aggregate a stream to a type or to a primitive: count(), max(), min(), and sum().
  - these operations work according to the predefined implementation.
- reduce() and collect() methods.--Customize a Stream's reduction mechanism

reduce() method

- 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.
- Reduction is simple, flexible, and parallelizable, and operates at a higher level of abstraction than imperative accumulation.
- Reduction (also known as folding) is a technique from functional programming that abstracts over many different accumulation operations.
- Given a nonempty sequence X of elements x1, x2, ..., xn of type T and a binary operator on T (represented here by *), the reduction of X under * is defined as:

(x1 * x2 * ... * xn)

reduce() method

- When applied to a sequence of numbers using ordinary addition as the binary operator, reduction is simply summation.
- But many other operations can be described by reduction.
- If the binary operator is "take the larger of the two elements" (which could be represented in Java by the lambda expression **(x,y) -> Math.max(x,y)**, or more simply as the method reference **Math::max**), reduction corresponds to finding the maximal value.
- Useful todescribe the computation in a more abstract and compact way, as well as a more parallel-friendly way--provided your binary operator satisfies a simple condition: associativity.

reduce() method

- A binary operator * is associative if, for any elements a, b, and c:
- ((a * b) * c) = (a * (b * c))
- Associativity means that grouping doesn't matter.
- If the binary operator is associative, the reduction can be safely performed in any order.
- In a sequential execution, the natural order of execution is from left to right;
- In a parallel execution, the data is partitioned into segments, reduce each segment separately, and combine the results.
- Associativity ensures that these two approaches yield the same answer. This is easier to see if the definition of associativity is expanded to four terms:
- (((a * b) * c) * d) = ((a * b) * (c * d))

reduce() Method

- Three variations of this method, which differ by their signatures and returning types.
- Can have the following parameters:
- **identity –** the initial value for an accumulator, or a default value if a stream is empty and there is nothing to accumulate
- **accumulator –** a function which specifies the logic of the aggregation of elements.
  - As the accumulator creates a new value for every step of reducing, the quantity of new values equals the stream's size and only the last value is useful.
  - This is not very good for the performance.
- **combiner –** a function which aggregates the results of the accumulator.
  - Only call combiner in a parallel mode to reduce the results of accumulators from different threads.

reduce() Method

```
OptionalInt reduced = IntStream.range(1, 4).reduce((a, b) -> a + b);
```

reduced = 6 (1 + 2 + 3)

```
int reducedTwoParams = IntStream.range(1, 4).reduce(10, (a, b) -> a + b);
```

reducedTwoParams = 16 (10 + 1 + 2 + 3)

```
int reducedParams = Stream.of(1, 2, 3) .reduce(10, (a, b) -> a + b, (a, b)
-> { log.info("combiner was called"); return a + b; });
```

reducedParams = 16 (10 + 1 + 2 + 3)

The result will be the same as in the previous example (16), and there will be no log info, which means that combiner wasn't called. **To make a combiner work, a stream should be parallel:**

reduce() Method

```
int reducedParallel = Arrays.asList(1, 2, 3).parallelStream() .reduce(10,
(a, b) -> a + b, (a, b) -> { log.info("combiner was called"); return a + b;
});
```

- Result here is different (36), and the combiner was called twice.
- Reduction works by the following algorithm: the accumulator ran three times by adding every element of the stream to *identity*.
- These actions are being done in parallel.
- As a result, they have (10 + 1 = 11; 10 + 2 = 12; 10 + 3 = 13;).
- Now combiner can merge these three results. It needs two iterations for that (12 + 13 = 25; 25 + 11 = 36).

reduce() Method

```
Comparator<Person> byHeight = Comparators.comparingInt(Person::getHeight);
BinaryOperator<Person> tallerOf = BinaryOperator.maxBy(byHeight);
Optional<Person> tallest = people.stream().reduce(tallerOf);
```

collect() Method

- reduction of a stream can also be executed by another **terminal** operation, the *collect()* method.
- Accepts an argument of the type *Collector Interface*, which specifies the mechanism of reduction.
- Already created, there exists predefined collectors for most common operations.
- Can be accessed with the help of the *Collectors* type.
- For some not trivial tasks, a custom Collector can be created
- Performs 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

Collectors

- All predefined implementations can be found in the Collectors class.
- Can use the following static import with them to leverage increased readability:

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

- Can also use single import collectors of choice

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

```java
class Product{
    private String name;
    private int price;
    @Override
    public String toString() {
        return "Product [name=" + name + "]";
    }
    public int getPrice() {
        return price;
    }
    public void setPrice(int price) {
        this.price = price;
    }
    public Product(int price, String name) {
        this.price = price;
        this.name = name;
    }
    public String getName()
    {
        return name;
    }
}
```

collect() Method

```
List<Product> productList = Arrays.asList(new Product(23, "potatoes"), new
Product(14, "orange"), new Product(13, "lemon"), new Product(23, "bread"),
new Product(13, "sugar"));
```

Converting a stream to the *Collection* (*Collection*, *List* or *Set*):

```
List<String> collectorCollection =
productList.stream().map(Product::getName).collect(Collectors.toList());
```

collect() Method

```
List<Product> productList = Arrays.asList(new Product(23, "potatoes"), new
Product(14, "orange"), new Product(13, "lemon"), new Product(23, "bread"),
new Product(13, "sugar"));
```

Reducing to *String*:

```
String listToString = productList.stream().map(Product::getName)
.collect(Collectors.joining(", ", "[", "]"));
```

- *joining()* method can have from one to three parameters (delimiter, prefix, suffix).
- Most convenient thing about using *joining()* is that the developer doesn't need to check if the stream reaches its end to apply the suffix and not to apply a delimiter.
- *Collector* will take care of that

collect() Method

```
List<Product> productList = Arrays.asList(new Product(23, "potatoes"), new
Product(14, "orange"), new Product(13, "lemon"), new Product(23, "bread"),
new Product(13, "sugar"));
```

Processing the average value of all numeric elements of the stream:

```
double averagePrice = productList.stream()
.collect(Collectors.averagingInt(Product::getPrice));
```

collect() Method

```
List<Product> productList = Arrays.asList(new Product(23, "potatoes"), new
Product(14, "orange"), new Product(13, "lemon"), new Product(23, "bread"),
new Product(13, "sugar"));
```

Processing the sum of all numeric elements of the stream:

```
int summingPrice = productList.stream()
.collect(Collectors.summingInt(Product::getPrice));
```

- methods *averagingXX()*, *summingXX()* and *summarizingXX()* can work with primitives (*int*, *long*, *double*) and with their wrapper classes (*Integer*, *Long*, *Double*).
- One more powerful feature of these methods is providing the mapping.
- As a result, the developer doesn't need to use an additional *map()* operation before the *collect()* method.

collect() Method

```
List<Product> productList = Arrays.asList(new Product(23, "potatoes"), new
Product(14, "orange"), new Product(13, "lemon"), new Product(23, "bread"),
new Product(13, "sugar"));
```

Collecting statistical information about stream's elements:

```
IntSummaryStatistics statistics = productList.stream()
.collect(Collectors.summarizingInt(Product::getPrice));
```

- By using the resulting instance of type *IntSummaryStatistics*, the developer can create a statistical report by applying the *toString()* method.
- The result will be a *String* common to this one *"IntSummaryStatistics{count=5, sum=86, min=13, average=17,200000, max=23}."*
- Also easy to extract from this object separate values for count, sum, min, average, and max by applying the methods getCount(), getSum(), getMin(), getAverage(), and getMax(). All of these values can be extracted from a single pipeline.

collect() Method

```
List<Product> productList = Arrays.asList(new Product(23, "potatoes"), new
Product(14, "orange"), new Product(13, "lemon"), new Product(23, "bread"),
new Product(13, "sugar"));
```

Grouping of stream's elements according to the specified function:

```
Map<Integer, List<Product>> collectorMapOfLists = productList.stream()
.collect(Collectors.groupingBy(Product::getPrice));
```

- stream was reduced to the *Map*, which groups all products by their price

collect() Method

```
List<Product> productList = Arrays.asList(new Product(23, "potatoes"), new
Product(14, "orange"), new Product(13, "lemon"), new Product(23, "bread"),
new Product(13, "sugar"));
```

Dividing stream's elements into groups according to some predicate:

```
Map<Boolean, List<Product>> mapPartioned = productList.stream()
.collect(Collectors.partitioningBy(element -> element.getPrice() >
15));
```

- 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

collect() Method

```
List<Product> productList = Arrays.asList(new Product(23, "potatoes"), new
Product(14, "orange"), new Product(13, "lemon"), new Product(23, "bread"),
new Product(13, "sugar"));
```

Pushing the collector to perform additional transformation:

```
Set<Product> unmodifiableSet = productList.stream()
.collect(Collectors.collectingAndThen(Collectors.toSet(),
Collections::unmodifiableSet));
```

- Collector has converted a stream to a *Set*, and then created the unchangeable *Set* out of it.

Collectors.teeing()

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

```
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
 ))
```

collect() Method

Custom collector:
If for some reason a custom collector should be created, the easiest and least verbose way of doing so is to use the method *of()* of the type *Collector*.

```
Collector<Product, ?, LinkedList<Product>> toLinkedList =
Collector.of(LinkedList::new, LinkedList::add, (first, second) -> {
first.addAll(second); return first; });

LinkedList<Product> linkedListOfProduct =
productList.stream().collect(toLinkedList);
```

In this example, an instance of the *Collector* got reduced to the *LinkedList*<Product>.

Collectors class

Collector	Behavior
toList()	Collect the elements to a List.
toSet()	Collect the elements to a Set.
toCollection(Supplier)	Collect the elements to a specific kind of Collection.
toMap(Function<T, K>, Function<T, V>)	Collect the elements to a Map, transforming the elements into keys and values according to the provided mapping functions.
summingInt(ToIntFunction<T>)	Compute the sum of applying the provided int-valued mapping function to each element (also versions for long and double).
summarizingInt(ToIntFunction<T>)	Compute the sum, min, max, count, and average of the results of applying the provided int-valued mapping function to each element (also versions for long and double).

Collectors class

Collector	Behavior
reducing()	Apply a reduction to the elements (usually used as a downstream collector, such as with groupingBy) (various versions).
partitioningBy(Predicate<T>)	Divide the elements into two groups: those for which the supplied predicate holds and those for which it doesn't.
partitioningBy(Predicate<T>, Collector)	Partition the elements, and process each partition with the specified downstream collector.
groupingBy(Function<T,U>)	Group elements into a Map whose keys are the provided function applied to the elements of the stream, and whose values are lists of elements that share that key.
groupingBy(Function<T,U>, Collector)	Group the elements, and process the values associated with each group with the specified downstream collector.
minBy(BinaryOperator<T>)	Compute the minimal value of the elements (also maxBy()).

Collectors class

Collector	Behavior
mapping(Function<T,U>, Collector)	Apply the provided mapping function to each element, and process with the specified downstream collector (usually used as a downstream collector itself, such as with groupingBy).
joining()	Assuming elements of type String, join the elements into a string, possibly with a delimiter, prefix, and suffix.
counting()	Compute the count of elements. (Usually used as a downstream collector.)

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 listOfNumbers = Arrays.asList(1, 2, 3, 4);

 listOfNumbers.stream().forEach(number -> System.out.println(number + " " +
Thread.currentThread(). getName()) );
```

- Output:

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

Parallel Streams

- Any stream in Java can easily be transformed from sequential to parallel.
- 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 listOfNumbers = Arrays.asList(1, 2, 3, 4);
listOfNumbers.parallelStream().forEach(number -> System.out.println(number + " " + Thread.currentThread(). getName())
);
```

- Execute code in parallel on separate cores.
- Final result is the combination of each individual outcome.
- Order of execution is out of our control.
- 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

- API allows to create parallel streams, which perform operations in a parallel mode.
- Stream API automatically uses the *ForkJoin* framework to execute operations in parallel.
- By default, the common thread pool will be used and there is no way (at least for now) to assign some custom thread pool to it.
- Can be overcome by using a custom set of parallel collectors.
- When using streams in parallel mode, avoid blocking operations.
- Also best to use parallel mode when tasks need a similar amount of time to execute.
- If one task lasts much longer than the other, it can slow down the complete app's workflow.

Parallel Streams

- When the source of a stream is a *Collection* or an *array*, it can be achieved with the help of the ***parallelStream()*** method:

```
Stream<Product> streamOfCollection = productList.parallelStream();
boolean isParallel = streamOfCollection.isParallel();
boolean bigPrice = streamOfCollection
  .map(product -> product.getPrice() * 12)
  .anyMatch(price -> price > 200);
```

Parallel Streams

If the source of a stream is something other than a *Collection* or an *array*, the ***parallel()*** method should be used:

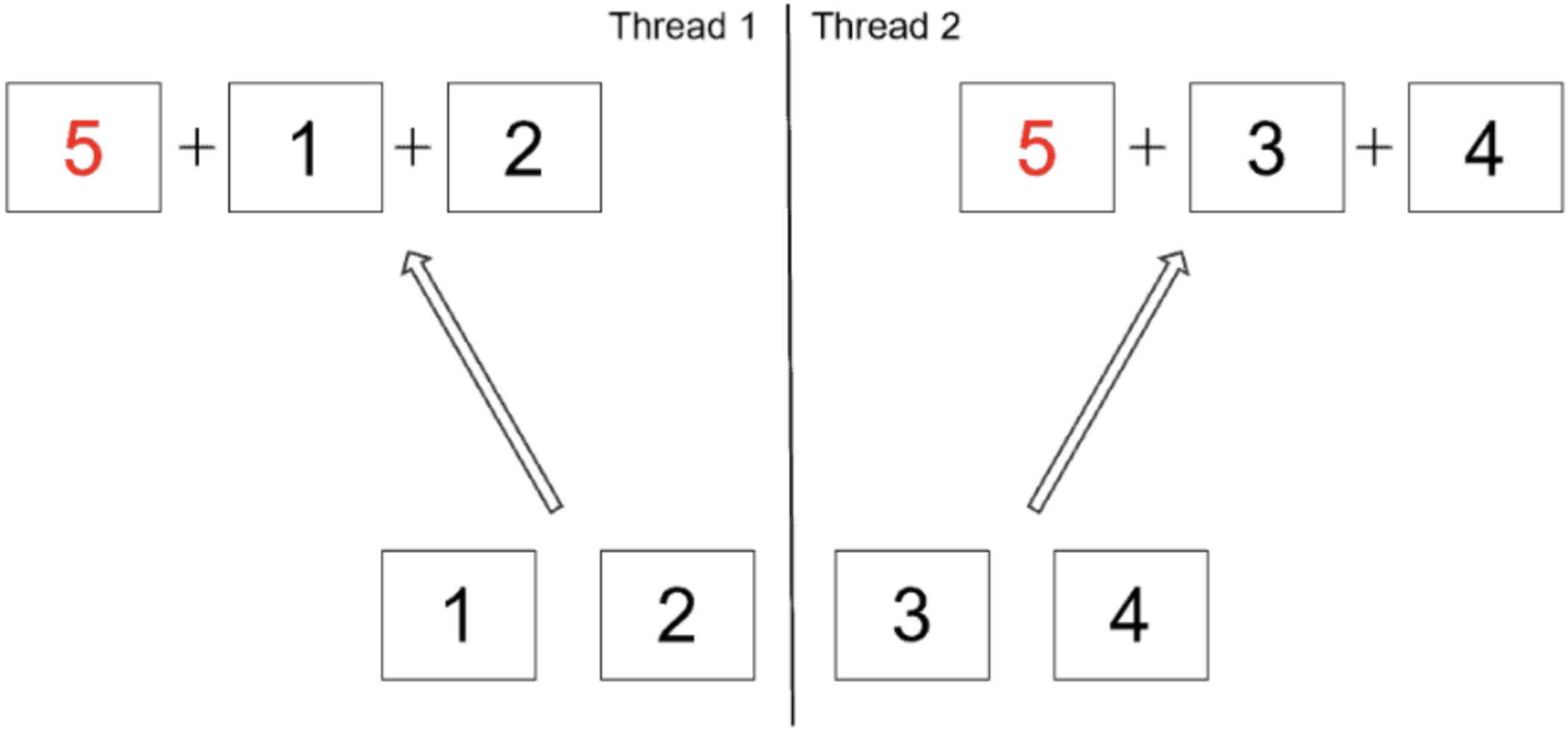
```
IntStream intStreamParallel = IntStream.range(1, 150).parallel();
boolean isParallel = intStreamParallel.isParallel();
```

Splitting Source

- 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.
- Make use of the reduce method and add five to the starting sum, instead of starting from zero

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

- 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



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

Parallel Streams

- In order to fix this issue, the number five should be added outside of the parallel stream
- Number of threads in the common pool is equal to (the number of processor cores -1)

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

Custom Thread Pool

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

```
List 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();
```

Performance Implication of parallel stream

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

Overhead in parallel streams

- On running a benchmark on a sequential and parallel reduction operation

```
IntStream.rangeClosed(1, 100).reduce(0, Integer::sum);
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
- Reason behind this is that sometimes the overhead of managing threads, sources, and results is a more expensive operation than doing the actual work

```
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
```

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 arrayListOfNumbers = new ArrayList<>();

private static final List linkedListOfNumbers = new LinkedList<>();

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

    linkedListOfNumbers.add(i);

}); }
```

Splitting Costs

- 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);
```

- 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.

Splitting Costs

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

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

```
arrayListOfNumbers.stream().reduce(0, Integer::sum);
arrayListOfNumbers.stream().parallel().reduce(0, Integer::sum);
arrayListOfNumbers.stream().collect(Collectors.toSet());
arrayListOfNumbers.stream().parallel().collect(Collectors. toSet())
```

Merging Costs

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

```
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
```

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

Memory Locality

- using two arrays, one using a primitive type, and the other using an object data type

```
private static final int[] intArray = new int[1_000_000];
private static final Integer[] integerArray = new Integer[1_000_000];
static {
    IntStream.rangeClosed(1, 1_000_000)
    .forEach(i -> {
        intArray[i-1] = i;
        integerArray[i-1] = i;
     }); }
```

Memory Locality

- Benchmark on a sequential and parallel reduction operation on the two arrays
- Arrays.stream(intArray).reduce(0, Integer::sum);
  Arrays.stream(intArray).parallel().reduce(0, Integer::sum);
  Arrays.stream(integerArray).reduce(0, Integer::sum);
  Arrays.stream(integerArray).parallel().reduce(0, Integer::sum);

```
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
```

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

- 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

When to Use Parallel Streams

- 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.