Mike Burton

A great lab, many thanks!

I deviated slightly, and calculated total of lengths of words in file:

First try, using reduce "worked" but is calculating Length of (sum of strings)

```
n = br.lines()
  .flatMap(l -> Arrays.stream(l.split(" ")))
  .reduce("", (String i, String j) -> i+j )
  .length();
```

Then did a better approach, sum of (lengths of the strings)

```
n = br.lines()
  .flatMap(l -> Arrays.stream(l.split(" ")))
  .map(s -> s.length())
  .sum();
```

Also some feedback re NetBeans on Mac (feel free to forward or point me to a better place to post if nec)
File-chooser isn't well integrated, eg can't drag a folder from Finder into it.
Clyppan: Couldnt paste previous "copy"s , had to additionally press CMD+V every time.

Stephen Colebourne

This works:

```
 ToIntFunction<String> ref = String::length;
 strs.sort(Comparators.comparing(ref).thenComparing(Comparators.naturalOrder()));
```

This doesn't work:

```
 strs.sort(Comparators.comparing(String::length).thenComparing(Comparators.naturalOrder()));
```

Nor does this work:

```
 strs = strs.stream().sorted(Comparators.comparing(String::length)).sorted().collect(Collectors.toList());
```

This works:

```
 strs.sort(Comparators.comparing((String s) -> s.length()).thenComparing(Comparators.naturalOrder()));
```

This doesn't work:

```
 strs.sort(Comparators.comparing(s -> s.length()).thenComparing(Comparators.naturalOrder()));
```

Nor does this work:

```
 strs = strs.stream().sorted(Comparators.comparing(s -> s.length())).sorted().collect(Collectors.toList());
```

I tried:

```
 strs = strs.stream().sorted(comparing((String s) -> s.length())).sorted().collect(toList());
```

and thought it would sort by length, then natural order within length, but (of course) it didn't.
This may well be quite a common mistake.

flatMap() has five forms. Four are based on FlatMapper interface, one is based on Function. In the IDE (NetBeans & IntelliJ), the higlghted interface names are FlatMapper, Function, ToInt, ToDouble and ToLong. We spent five minutes looking at ToInt/ToLong/ToDouble assuming they were overrides of Function, when they are in fact overrides of FlatMapper. I *really* think that interface names IntFlatMapper, LongFlatMapper and DoubleFlatMapper would be *much* more obvious to use in the IDE.

When trying to use FlatMapper it needed a "Consumer", so I looked for a "Consumers" class but didn't find one.

I tried:
a tutorial for lambda: http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
but it was using Blocks in the last example and Blocks are no longer supported!!

Feedback from Edward Wong
> Kind of confusing how to get a collection back out of my Stream. The collect method is not obvious and the type inference seems a bit poor. For instance:

```
        List<Object> sortedList = stringList.stream()
                .sorted((s1, s2) -> Integer.compare(s1.length(), s2.length()))
                .collect(Collectors.toList());
```

The default Collector gives an Object stream, which I suppose is fair... If I specify the type I can get back what I want:

```
        List<String> sortedList = stringList.stream()
                .sorted((s1, s2) -> Integer.compare(s1.length(), s2.length()))
                .collect(Collectors.<String>toList());
```

Perhaps what would be better would be something like:

```
        List<String> sortedList = stringList.stream()
                .sorted((s1, s2) -> Integer.compare(s1.length(), s2.length()))
                .toList();
```

FOR RICHARD W:

```
public static void main(String... ignored) {
        List<String> stringList = new LinkedList<>();
        stringList.add("d");
        stringList.add("aa");
        stringList.add("a");
        stringList.add("c");
        stringList.add("b");

        System.out.println("Unsorted: " + stringList);

        Comparator<? super String> lengthComparator = (s1, s2) -> Integer.compare(s1.length(), s2.length());

        Comparator<String> lengthThenAlphabeticSorter
                = lengthComparator.thenComparing(String::compareTo);

        List<String> sortedList = stringList.stream()
                .sorted(lengthThenAlphabeticSorter)
                .collect(Collectors.<String>toList());

        System.out.println("Sorted: " + sortedList);
    }

    // input [d, aa, a, c, b]
// output [a, b, c, d, aa]
```

We were discussing how to inline all of this :)

Further problems with above code as well, running will give following error:

java: reference to thenComparing is ambiguous

both method <S>thenComparing(java.util.function.ToLongFunction<? super S>) in java.util.Comparator and method <S>thenComparing(java.util.function.ToDoubleFunction<? super S>) in java.util.Comparator match

Hence the IDE casted the String::compareTo :

```
Comparator<String> lengthThenAlphabeticSorter
                = lengthComparator.thenComparing((Comparator<String>) String::compareTo);
```

> Implementing sum for command line args

My initial attempt was to write this:

```
    private static void printOutSumOfArgs(String... args) {
        Collection<BigDecimal> arguments = new ArrayList<>(args.length);

        for (String arg : args) {
            arguments.add(new BigDecimal(arg));
        }

        BinaryOperator<BigDecimal> addingReducer = (x, y) -> x.add(y);
        BigDecimal sum = arguments.stream().reduce(BigDecimal.ZERO, addingReducer);
        System.out.println("The sum of " + arguments + " is: " + sum);
    }
```

However when I tried to refactor and inline to the following:

```
    private static void printOutSumOfArgs(String... args) {
        Collection<BigDecimal> arguments = new ArrayList<>(args.length);

        for (String arg : args) {
            arguments.add(new BigDecimal(arg));
        }

        BigDecimal sum = arguments.stream()
                .reduce(BigDecimal.ZERO, BigDecimal::add);
        System.out.println("The sum of " + arguments + " is: " + sum);
    }
```

IntelliJ reports an error (BinaryOperator cannot be applied to method reference). [Minor concern, IntelliJ’s problem].

More feedback:

We found a new method:

```
    private static final ToDoubleFunction toDoubleFunction = d -> (double) d;

    private static void printOutSumOfArgs(Collection<Double> arguments) {
        Double sum = arguments.stream()
                .map(toDoubleFunction)
                .sum();
        System.out.println("The sum of " + arguments + " is: " + sum);
    }
```

What is a bit counter-intuitive is the need to map a boxed Stream to its unboxed version. Is there a more elegant solution to this?

Steven Van Impe

Wanted to process as follows
- create stream of String
- convert each String to stream of char
- merge all the streams of char into one stream of char
- then process further

Couldn’t work out how to merge the streams of char into a single stream of char

---

John Oliver

It seems to be a common operation to go from Stream<Integer> to IntStream so you can use sum, average etc. But there appears to be no common/easy way to go between the two.

Stephen Colebourne
My experience in actually trying to get the word frequency test working :-(
It was a question of counting the frequency of words in a file.

```
public class CountWordFreq {

	public static void main(String[] args) throws Exception {
	try (BufferedReader br = new BufferedReader(new
InputStreamReader(CountWordFreq.class.getResourceAsStream("book.txt")))) {
		// apparently this is the right solution
		// the second argument wasn't obvious and the s->1 seems especially odd
		Map<String, Integer> right = br.lines().flatMap(s -> Arrays.stream(s.split(" ")))
			.collect(Collectors.groupingBy(s -> s, Collectors.reducing(s -> 1, Integer::sum)));
		System.out.println(right);
	}

	try (BufferedReader br = new BufferedReader(new
InputStreamReader(CountWordFreq.class.getResourceAsStream("book.txt")))) {
		// this is what I did on my own
		// I could get my stream of words, and then group them by word
		// but I could not convert the list of identical words to a count
		// or avoid creating the list of identical words in the first place
		// BTW, the s->s looked very weird and wrong, but I went with it
		Map<String, List<String>> initial = br.lines().flatMap(s -> Arrays.stream(s.split(" ")))
			.collect(Collectors.groupingBy(s -> s));

		// so given where I was the best solution seemed to be to do a second stream operation
		// but Map didn't have a stream(), so I had to use entrySet().stream()
		// that allowed me to map the list to its size easily emough
		// but converting back to a simple Map was a world of hurt
		// here is as far as I managed to get without a hint
		Map<Map.Entry<String, Integer>, Integer> freq1 = initial
			.entrySet().stream()
			.map(entry -> new AbstractMap.SimpleImmutableEntry<String, Integer>(entry.getKey(),
entry.getValue().size()))
```

```
                .collect(Collectors.toMap(entry -> entry.getValue()));
        System.out.println(freq1);

        // after being pointed at the three argument version of collect()
        // I managed to guess the three lambdas from the generics
        // but I had trouble with a compile error, so I had to assign each to variable
        // and eventually managed to get the right result
        Supplier<HashMap<String, Integer>> supplier = () -> new HashMap<String, Integer>();
        BiConsumer<HashMap<String, Integer>, Map.Entry<String, Integer>> accum =
                (HashMap<String, Integer> result, Map.Entry<String, Integer> entry) ->
result.put(entry.getKey(), entry.getValue());
        BiConsumer<HashMap<String, Integer>, HashMap<String, Integer>> merger = HashMap::putAll;
        Map<String, Integer> freq2 = initial
                .entrySet().stream()
                .map(entry -> new AbstractMap.SimpleImmutableEntry<String, Integer>(entry.getKey(),
entry.getValue().size()))
                .collect(supplier, accum, merger);
        System.out.println(freq2);
    }
    }

}
```