Victoria University of Wellington
School of Engineering and Computer Science

SWEN221: Software Development

Lab Handout

The purpose of this Tutorial is to sharpen your understanding of Java Lambda and Stream. Useful information can be found from

- `https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`,
- `https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html`.

There are three simple exercises, which can be completed independently.

1 FilterClass

Given a stream, one common operation is to filter only the ones of a certain type; this is usually obtained by the following code

```
List<Foo> res = data.stream()
        .filter(e -> e instanceof Foo)
        .map(e -> (Foo) e)
        .collect(Collectors.toList());
```

This is a little sub optimal, since we have to write two operations (filter and map) and we have to repeat the name of the class. It is possible to use `Stream.flatMap` and write a proper method `FilterClass.isInstanceOf(class)` so that

```
List<Foo> res = data.stream()
        .flatMap(FilterClass.isInstanceOf(Foo.class))
        .collect(Collectors.toList());
```

would have the same behaviour of the code shown before.

1.1 Stream.flatMap

`Stream.flatMap(Function mapper)` takes a stream, and replaces each element of this stream with the content of a mapped stream produced by applying the `mapper` function to each element.

For example, the following code output all the positive numbers twice, and filter out all the negatives:

```
myList.stream().flatMap(
  i -> {
    if (i >= 0) { return Stream.of(i, i); }
    return Stream.empty();
    }).collect(Collectors.toList());
```

Details of `Stream.flatMap` can be found in `https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#flatMap-java.util.function.Function-`.

Your task is to complete the implementation of the method `FilterClass.isInstanceOf(class)`.

You should now see the correlated tests passing.

2 Simplify reflection for invoking a method

Using reflection for invoking a method may fill our code with repetitive `try-catch`, as in the iconic code here:

```
try {/*example*/
  return String.class.getMethod("toString").invoke("");
}
catch (IllegalAccessException | NoSuchMethodException e) {
  throw new Error("Unexpected shape of the classes", e);
}
catch (SecurityException e) {
  throw new Error("Reflection blocked by security manager", e);
}
catch (InvocationTargetException e) {
  Throwable cause = e.getCause();
  if (cause instanceof RuntimeException) {throw (RuntimeException) cause;}
  if (cause instanceof Error) {throw (Error) cause;}
  throw new Error("Unexpected checked exception", cause);
}
```

This boring code usually has to be repeated many times in projects using reflections. With lambdas, it is possible to parametrise the above code with the following syntax:

```
return ReflectionHelper.tryCatch(
  () -> String.class.getMethod("toString").invoke("")
);
```

where the `reflection` method takes the body of the `try`, and execute it inside a correct `try-catch` .

Your task is to complete the implementation of the method `ReflectionHelper.tryCatch()`, so it parametrises the `try-catch` for invoking a method with reflection.

You should now see the correlated tests passing.

Note: The `tryCatch` method should contain the same error handling behaviour as shown in the handout.

3 Counting words

Here we use parallel streams to count how many times a specific word occurs into a document. This exercise require to use the 3-argument `Stream.reduce` variant

`reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)`.

The method is used for processing the `reduce` operation in parallel. Each sub-stream is processed by `accumulator`. Then, the results of the sub-streams are combined by `combiner`.

For example, in order to calculate $2*2+2*3+2*4$, one can partition the stream $(2,3,4)$ into three sub-streams, calculating $2*2$, $2*3$ and $2*4$, that is, element times 2. In other words, the initial value of the sub-stream is 2, and the result is obtained by multiplication. Finally, the results of the sub-streams are summed up. The code is written as follows:

```
List<Integer> list = Arrays.asList(2, 3, 4);
  int res = list.parallelStream()
        .reduce(2, (s1, s2) -> s1 * s2, (p, q) -> p + q);
```

Useful information of stream reduction can be found from

- `https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html`.
- `https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#reduce-U-java.util.function.BiFunction-java.util.function.BinaryOperator-`.

You can find two partially implemented versions that use parallel streams. In one we process the file line by line, in the other we process the file as a single stream of words. Your task is to complete the implementation of the methods

- `KeywordFinder.count1(String, Path)` and
- `KeywordFinder.count2(String, Path)`.

In a correct implementation, they will have the same behaviour.

You should now see the correlated tests passing.

Submission

Your lab solution should be submitted electronically via the *online submission system*, linked from the course homepage. The required files are:

```
swen221/lab9/flatMapFilterClass1/FilterClass.java
swen221/lab9/flatMapFilterClass1/FilterClassTest.java
swen221/lab9/simplyfyReflection2/ReflectionHelper.java
swen221/lab9/simplyfyReflection2/ReflectionHelperTest.java
swen221/lab9/parallelStream3/KeywordFinder.java
swen221/lab9/parallelStream3/KeywordFinderTest.java
```

You must ensure your submission meets the following requirements (which are needed for the automatic marking script):

1. **Your submission is packaged into a jar file, including the source code**. *Note, the jar file does not need to be executable.* See the following Eclipse tutorials for more on this:

`http://ecs.victoria.ac.nz/Support/TechNoteEclipseTutorials`

2. **The names of all classes, methods and packages remain unchanged**. That is, you may add new classes and/or new methods and you may modify the body of existing methods. However, you may not change the name of any existing class, method or package. *This is to ensure the automatic marking script can test your code.*

3. **All JUnit test files supplied for the assignment remain unchanged.** Specifically, you cannot alter the way in which your code is tested as the marking script relies on this. This does not prohibit you from adding new tests, as you can still create additional JUnit test files. *This is to ensure the automatic marking script can test your code.*

4. **You have removed any debugging code that produces output, or otherwise affects the computation.** *This ensures the output seen by the automatic marking script does not include spurious information.*

Note: Failure to meet these requirements could result in your submission being reject by the submission system and/or zero marks being awarded.