Victoria University of Wellington
School of Engineering and Computer Science

SWEN221: Software Development

Lab Handout

1 overview

The purpose of this Tutorial is to sharpen your understanding of Java Lambda. There are three simple exercises, that can be completed independently.
The first exercise uses Stream.flatMap. Both map and flatMap can be applied to a Stream<T> and they both return a Stream<R>; The difference between Stream.map and Stream.flatMap is that the *map lambda* produces one output value for each input value, whereas the *flatMap lambda* produces an arbitrary number (zero or more) values for each input value. This is obtained by making the *flatmap lambda* returning a Stream<R> result. Then flatmap itself takes all those streams and condenses them down to a single stream.

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

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

2 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 flatMap and write an opportune 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.

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

You should now see the correlated tests passing.

3 Simplify reflection

Using reflection may fill our code with repetitive try-catching, 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);
}
```

Often you know that either you are doing reflection right, or that if its wrong is a bug, so you want to catch checked exceptions and throw them as errors, with some special case, like for InvocationTargetException. This boring code usually have to be repeated many times in projects using reflections. With lambdas is possible to parametrize the code, and to obtain such a syntax:

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

Where the reflection method takes the body of the try, and execute it inside of a correct try−catch

Your task is to complete the implementation of the method ReflectionHelper.reflection(..).

You should now see the correlated tests passing.

Note well: The tryCatch method should contain the error handling behaviour shown in the handout.

4 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 variant of Stream.reduce that takes 3 arguments. See `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 as 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/lambda/flatMapFilterClass1/FilterClass.java
swen221/lambda/flatMapFilterClass1/FilterClassTest.java
swen221/lambda/simplyfyReflection2/ReflectionHelper.java
swen221/lambda/simplyfyReflection2/ReflectionHelperTest.java
swen221/lambda/parallelStream3/KeywordFinder.java
swen221/lambda/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.