Lab 10

1. *Custom Annotations.* In this problem, you will use an expanded version of the custom annotation `@BugReport` discussed in the slides to create a small bug-reporting tool. The `@BugReport` annotation has been expanded for you to include two new elements:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface BugReport {
    String assignedTo() default "<unassigned>";
    int severity() default 0;
    String description() default "";
    String reportedBy() default "<unnamed>";

}
```

This annotation, together with start-up code for the reporting tool `BugReportGenerator` and a `Main` class, can be found in the package `lesson10.labs.prob2.bugreporter`. You will need to complete the code in the `BugReportGenerator`, according to the specifications below.

Instances of the annotation have been placed at the class level in each of the classes in the package `lesson10.labs.prob2.javapackage`, in order to indicate problems that need to be fixed in each of these classes, together with names of the individuals assigned to make the bugfixes. For instance:

```
@BugReport(assignedTo="Tom Jones", reportedBy="Corazza", description="computePerimeter incorrect")
public class Circle implements ClosedCurve {
    private double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    public double getRadius() {
        return radius;
    }
    public void setRadius(double radius) {
        this.radius = radius;
    }

    @Override
    public double computePerimeter() {
        return Math.PI * radius * radius;
    }
}
```

The method `reportGenerator` in the `BugReportGenerator` class should do the following:

(1) Form a list of all classes in the package `lesson10.labs.prob1.javapackage`
(2) For each class in the package, extract the bug report information supplied by the elements of the `@BugReport` annotation

(3) Create a report that indicates the list of bugs (with detailed information) that is assigned to each bugfixer (format shown below)
(4) Output the report to a file `bug_report.txt`.

For (1), a method `ClassFinder.find(`**`PACKAGE_TO_SCAN`**`)` has been provided for you already; it extracts a list of `Class`es from a given package; the source for this method can be found in `lesson10.labs.prob1.classfinder`; it does not need to (and should not be) modified.

For (4), you will need to use a FileWriter (together with a PrintWriter) to output a file. Use the try-with-resources mechanism for creating an instance of these Writers.

When your code is complete and the `main` method of `Main` is run, the output file `bug_report.txt` should look like this:

```
Tom Jones
  reportedBy: Corazza
  classname: lesson10.labsolns.prob2.javapackage.Circle
  description: computePerimeter incorrect
  severity: 0

  reportedBy: Corazza
  classname: lesson10.labsolns.prob2.javapackage.ClosedCurve
  description:
  severity: 1

Joe Smith
  reportedBy: Corazza
  classname: lesson10.labsolns.prob2.javapackage.DataMiner
  description: Should use Logger
  severity: 1

  reportedBy: Corazza
  classname: lesson10.labsolns.prob2.javapackage.Rectangle
  description: computePerimeter incorrect
  severity: 2
```

2. In the package lesson10.labs.prob2, the file `OldFileIO.java` is designed to write a file called "output.txt" into this package. `OldFileIO` illustrates the pre-Java 8 way of writing a file. Create another Java class `NewFileIO` that does the same thing, but with some modifications:
    a. You will use the try-with-resources construct to instantiate the necessary Writers (and to handle the closing step in the background).
    b. *Approach to determining the path for the output file*. Think about the best way to do this. Using a URL to specify a file on the classpath (as in InClass exercise) only works for reads. However, do you think it would ever be necessary to write a file to a location on the classpath? (It is not necessary to change the way this output location has been specified in OldFileIO – but if you can think of a better way to do it, then try it.)

3. In the package lesson10.labs.prob3, there is a class `FixThis` in which a stream `map` is called which accesses another method that throws an `Exception`. The code will not compile as it is written. Use one of the Java 8 exception-handling strategies to get the code to compile and run – create a new class `FixThisSoln` for this purpose. A (commented) `main` method is provided. Expected output for the first call to `processList` is

   `[not, too, big, yet]`

   However, the second call should throw a `RuntimeException`.

4. In the package `lesson10.labs.prob4`, there is a class `GuestListPreJava8` which includes a method for extracting (in sorted order) from a list of invited guests (for a particular event) all those guests who have said they will attend the event, who are female, and who are not "illegal." The implementation has been done using pre-Java 8 techniques. Your job in this exercise is to rewrite the primary method `printListOfExpectedFemaleGuests` by creating a `Stream` pipleline and using filters and maps, as necessary. Checking whether a guest is "illegal" involves a checked exception. You will need to use techniques discussed in the lecture to handle this. All the code you need has been provided for you; you only need to write code for the method `printListOfExpectedFemaleGuests.` Optionally, you may use the PredicateWithException functional interface that has been provided for you in working out your solution (similar to the use of FunctionWithException in the package lesson10.lecture.exceptions2); however, you are <u>not</u> required to solve the problem this way.

5. In the `package lesson10.labs.prob7`, there are classes `Main` and `Employee`. The `main` method in `Main` loads a list of `Employee` s and then attempts to print, in sorted order, the full names of those `Employee`s whose salary is greater than 100,000 and whose last name begins with any letter that comes after 'M' in the alphabet. This exercise asks you to refactor this processing step in the `main` method so that it can be unit tested, using the techniques mentioned in the Lesson. Do the following:
   a. It is difficult to test an expression that simply prints to console. Move this processing step into two methods, `asString(List)`, which does the same processing, but returns a `String` rather than printing to the console, and `printEmps(List)`, which calls `asString` and then prints the string to the console. Replace the processing step in the `main` method with a call to `printEmps`.
   b. Create two packages, `soln1`, `soln2`, where you will put the two different types of solutions you will develop for testing this code.
   c. In `soln1`, create a JUnit `Test` class that tests the `asString` method. Make sure you test with a few `Employee` instances so that at least one `Employee` is excluded from the list and at least one is included in the list. This is an example of the Simple approach mentioned in the slides.
   d. In `soln2`, refactor the `asString` method so that method references are used to call auxiliary methods, as in the Complex case described in the lecture. Create auxiliary methods `salaryGreaterThan100000(Employee e)` and `lastNameAfterM(Employee e)` for this purpose. Then create a `Test` class in `soln2`

that tests these auxiliary methods, along with the `fullName(Employee e)` method. Does this approach provide a good test for the `asString` method?

6. In the package there is a class Queue. Do the following:
    a. Show that Queue is not threadsafe by setting up a multithreaded environment in which you create a race condition.
    b. Modify Queue so that it is threadsafe, and verify in your test environment that you have been successful.