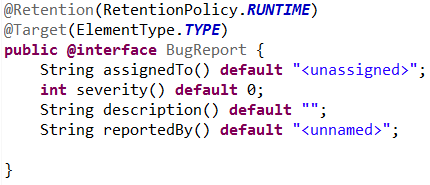
**Lab 10**

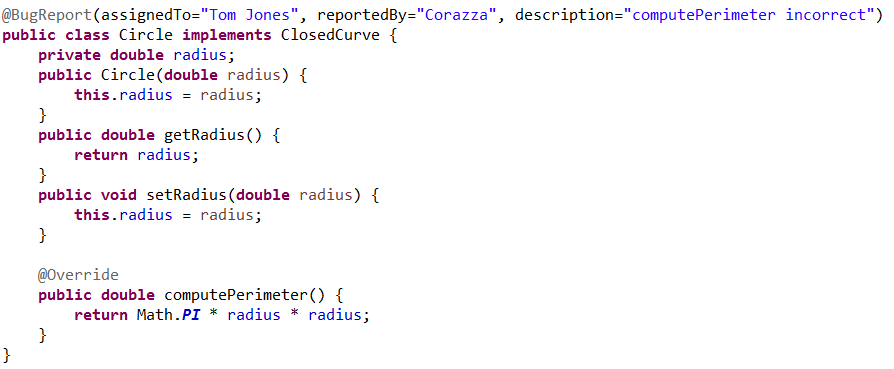
MPP Class July 2015: Do problems 1, 2, 5, 6, 7, 8 (and keep the numbering like this in your submission.

1. *TDD Practice.* Try developing code in a simple case using the TDD approach. For this problem, create a class TDDPractice and a method changeLastCharToUpper(List<String> words). The method should change the last letter of each String in the input list to upper case. At first, write a method that just returns an empty list. Then create a test class TestTDDPractice that has a method test (remember to annotate with @Test); the test method should perform a test to validate that your method works. When you run the test, it should fail initially (since you have not coded changeLastCharToUpper yet). Then write the code for changeLastCharToUpper, and test again.  
     
   Write a few comments about your experience doing this exercise. Does this approach seem useful?
2. *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:



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:

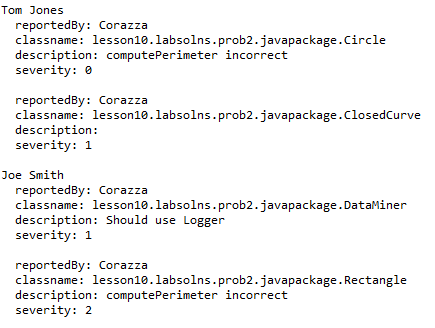


The method reportGenerator in the BugReportGenerator class should do the following:

1. Form a list of all classes in the package lesson10.labs.prob2.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 Classes from a given package; the source for this method can be found in lesson10.labs.prob2.classfinder; it does not need to (and should not be) modified.

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



1. In the package lesson10.labs.prob3, there are two files: OldFileIO.java and word\_test.txt . OldFileIO illustrates the pre-Java 8 way of reading a file and printing out the lines from the file. Create another Java class NewFileIO and do the same thing using a stream (use the lines() method from BufferedReader – read about this method in the API docs https://docs.oracle.com/javase/8/docs/api/). Do you need to handle the IOException in any special way?
2. Do the following operations involving the MPP database introduced in class.
   1. Implement the following method, provided in the code for this problem (requirements are below):  
       void updateZips()  
      A government survey requires you to update your Customer table in the following way: For those Customers whose 5-digit zipcode begins with the digit 5, append to the zip code the +4 code “1000”. For example, the Fairfield zip code 52556 must be changed to 52556-1000. One requirement is that it is important that either all zipcodes beginning with 5 have the +4 code, or none of them do; the data table will be corrupt if some do and some do not.  
        
      Think about how to retrieve the relevant zip codes. (The SQL command for computing the length of a string in JavaDB is LENGTH(str). The SQL command for trimming whitespace at the end of a string is RTRIM(str).) Think about how to do the updates. Which parts of the code should be placed within the bounds of a transaction?  
        
      Be sure to use the try-with-resources construct for closing of resources and exception handling.
   2. After you have completed part (a), create a method   
       List<Integer> insertCustomer(List<Customer> custs)  
      that will *insert* new Customer data into the Customer table. Your method should check the zip code of each customer on the list; whenever a 5-digit zip begins with the digit 5, replace the zip code with its +4 version (so, for example, the zip code 52556 would be replaced by 52556-1000). Pre-process the list in this way using a stream pipeline rather than a for loop. Perform the necessary INSERT actions to load the new customers into the Customer table, but do not INSERT a new row for a Customer if the Customer table already has a row with the same name and zip. Think about how a transaction should be used in this case. Keep track of the customer IDs for those new Customer s that have been inserted into the table, and return them in a collection of type List<Integer>.  
        
      For input data, use the main method provided for you.
3. In the package lesson10.labs.prob5, 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.

1. In the package lesson10.labs.prob6, 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.
2. 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 Employees 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:
   1. 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.
   2. Create two packages, soln1, soln2, where you will put the two different types of solutions you will develop for testing this code.
   3. 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.
   4. 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 lastNameAfterMEmployee 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?

8. In the package there is a class Queue. Do the following:

1. Show that Queue is not threadsafe by setting up a multithreaded environment in which you create a race condition.
2. Modify Queue so that it is threadsafe, and verify in your test environment that you have been successful.
3. (Extra Credit, 5 points). Redo Problem 5 using the data access framework provided. You will need to add the jar file dataaccess.jar to your project. To start, copy the code from lesson10.lecture.jdbc.framework and modify it as necessary.