**Lab 10**

1. In our public directory on cs5 in \Lecture10-code\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.

public static String asString(List<Employee> emps){  
 return emps.stream()  
 .filter(e -> (e.getSalary() > 100000))  
 .filter(e -> (e.getLastName().charAt(0) > 'M'))  
 .map(e -> *fullName*(e))  
 .sorted()  
 .collect(Collectors.*joining*(", "));  
}  
public static void printEmps(List<Employee> emps){  
 System.*out*.println(*asString*(emps));  
}

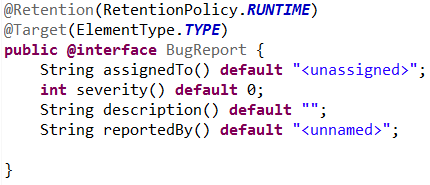
* 1. Create two packages, soln1, soln2, where you will put the two different types of solutions you will develop for testing this code.
  2. In soln1, create a JUnit Test class that tests the asString method. Make sure you test with a few Employee instances so that at least one Employee is excluded from the list and at least one is included in the list. This is an example of the Simple approach mentioned in the slides.

public class Tester {  
 @Test  
 public void testAsString(){  
 List<Employee> emps = Arrays.*asList*(new Employee("Joe", "Davis", 120000),  
 new Employee("John", "Sims", 110000),  
 new Employee("Joe", "Stevens", 200000),  
 new Employee("Andrew", "Reardon", 80000),  
 new Employee("Joe", "Cummings", 760000),  
 new Employee("Steven", "Walters", 135000),  
 new Employee("Thomas", "Blake", 111000),  
 new Employee("Alice", "Richards", 101000),  
 new Employee("Donald", "Trump", 100000));  
 String expectedResult = "Alice Richards, Joe Stevens, John Sims, Steven Walters";  
 *assertEquals*(Main.*asString*(emps),expectedResult);  
  
 }  
}

* 1. 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) andlastNameAfterMEmployee 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?

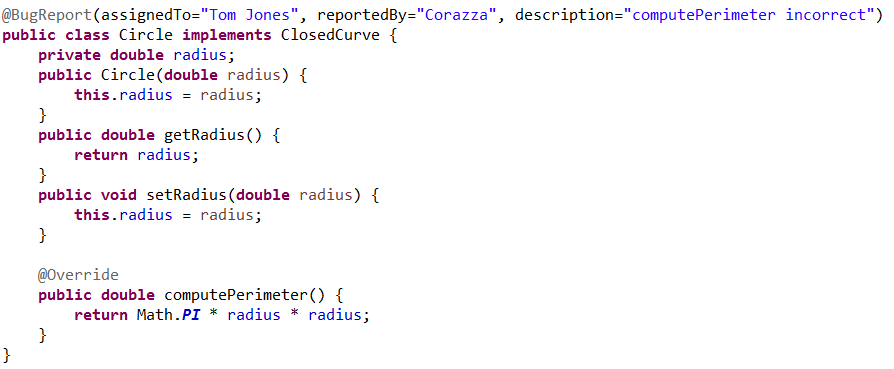
public class Tester {  
 @Test  
 public void testSalaryGreaterThan100000(){  
 *assertTrue*(Main.*salaryGreaterThan100000*(new Employee("Joe", "Davis", 120000)));  
 *assertFalse*(Main.*salaryGreaterThan100000*(new Employee("Joe", "Davis", 100000)));  
 *assertFalse*(Main.*salaryGreaterThan100000*(new Employee("Joe", "Davis", 90000)));  
 }  
 @Test  
 public void testLastNameAfterM(){  
 *assertFalse*(Main.*lastNameAfterM*(new Employee("Joe", "Davis", 90000)));  
 *assertTrue*(Main.*lastNameAfterM*(new Employee("Steven", "Walters", 135000)));  
 }  
 @Test  
 public void testFullName() throws NoSuchMethodException, InvocationTargetException, IllegalAccessException {  
 Method method = Main.class.getDeclaredMethod("fullName", Employee.class);  
 method.setAccessible(true);  
 String res=(String)method.invoke(new Main(), new Employee("Steven", "Walters", 135000));  
 String exceptRes="Steven Walters";  
 *assertEquals*(exceptRes,res);  
 }  
  
}

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:



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:

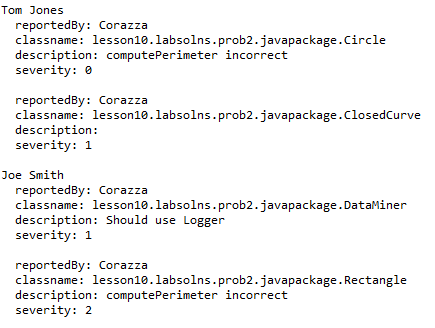


The method reportGenerator in the BugReportGenerator class should do the following:

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:



public class BugReportGenerator {  
 private static final Logger *LOG* = Logger.*getLogger*(BugReportGenerator.class.getName());  
 private static final String *PACKAGE\_TO\_SCAN* = "lesson10.labs.prob2.javapackage";  
 private static final String *REPORT\_NAME* = "bug\_report.txt";  
 private static final String *REPORTED\_BY* = " reportedBy: ";  
 private static final String *CLASS\_NAME* = " classname: ";  
 private static final String *DESCRIPTION* = " description: ";  
 private static final String *SEVERITY* = " severity: ";  
 public void reportGenerator() throws FileNotFoundException {  
 List<Class<?>> classes = ClassFinder.*find*(*PACKAGE\_TO\_SCAN*);  
 //implement  
 try( PrintWriter out = new PrintWriter( *REPORT\_NAME*) ){  
 classes.forEach(c->{BugReport a=c.getAnnotation(BugReport.class);  
 out.println(a.assignedTo());  
 out.println(*REPORTED\_BY*+a.reportedBy());  
 out.println(*CLASS\_NAME*+c.toString());  
 out.println(*DESCRIPTION*+a.description());  
 out.println(*SEVERITY*+a.severity());  
 }  
 );  
 }  
 }  
}

8. In the in our public directory in \Lecture10-code\ 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.

public static void main(String[] args) throws InterruptedException {  
  
 Runnable r=()->{  
 Main.*queue*.add(new Object());  
 };  
  
 List<Thread> threads=new ArrayList<>();  
 for (int i = 0; i < 1; i++) {  
 Thread t= new Thread(r);  
 threads.add(t);  
 t.start();  
 }  
   
 for(Thread t:threads){  
 t.wait();  
 }  
 // Thread.sleep(1000);  
  
 System.*out*.println(*queue*.size());  
}

1. Modify Queue so that it is threadsafe, and verify in your test environment that you have been successful.

public class Queue {  
 class Node {  
 Object value;  
 Node next;  
 }  
 private Node head;  
 private Node tail;  
 public int size(){  
 int size=0;  
 Node node=head;  
 while(node!=null){  
 ++size;  
 node=node.next;  
 }  
 return size;  
 }  
 synchronized public void add(Object newValue) {  
 Node n = new Node();  
 if(head == null) head = n;  
 else tail.next = n;  
 tail = n;  
 tail.value = newValue;  
 }  
 synchronized public Object remove() {  
 if(head == null) return null;  
 Node n = head;  
 head = n.next;  
 return n.value;  
 }  
}