**Lab 7**

1. Short Answer
2. In an earlier lesson, it was mentioned that Java’s ArrayList implements 6 interfaces and extends one class. What are they?

**Answer:**

* 1. Extends: AbstractList<E>
  2. Implements: Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Parts B – D of this Problem refer to code in package lesson7.labs.prob1, in which you are trying to remove duplicates from a List and then test that your output is correct. All three attempts to solve this problem are incorrect in some way (when you run the code, output message indicates that the procedure fails). Explain, in each case, what is wrong with the solution. Place each of your answers in a text file in the relevant package.

**Answer:**

* **Part B:** The function public boolean equals(Employee e) is not an Overridding of equals(Object o) of Object class. When we use contain() method in class List, it loop though the elements in List and compare using equals(Object o) method from Object class (because JVM doesn’t find any overridding version) and this equals() method check 2 things:
  + obj1 == obj2
  + references of these two reference variable point to the same object
* so the result is false.
* **Part C:** Because Employee is used as a key and The Employee class doesn’t override hashCode() method so it is wrong when adding Employee object to the HashMap, each Employee object has different hash value.

**Part D:** Because this block of code

else {

tracker.get(e).setVisited(true);

}

When we found the object Employee in the HashMap then we set the *visited* attribute of this object to *true*. So this code tracker.containsKey(e) will no longer return true with the object has same *name, salary* but the visited is *false*

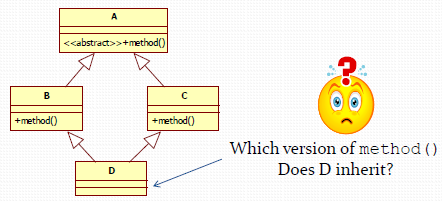
E. Lesson 5 introduced the Diamond Problem that must be handled by any language that supports multiple inheritance. Java SE 8 now supports “behavioral” multiple inheritance (but not “data” multiple inheritance). Explain how features of Java 8 handle the Diamond Problem by considering two scenarios:

1. When the type D is a class

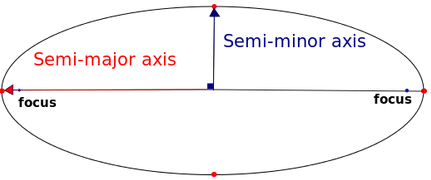
**Answer:** We must override the method method() in D or declare it as an abstract method.

ii. When the type D is an interface.

**Answer:** We must declare a method (unimplemented) in D or provide a default method of this method.



1. The Lesson 5 Demo in lesson5.lecture.intfaces2 shows how to polymorphically compute the average perimeter of a list of geometric objects by requiring each to implement the ClosedCurve interface. Notice that when a closed curve happens to be a polygon, computing the perimeter is especially easy – you just add up the lengths of the sides.   
     
   If we create an interface Polygon having method double[] getSides() (which will return the length of each side of the polygon in an array), we could replace ClosedCurve in our example with Polygon – *if* we didn’t have to take into account the computation of the perimeter of *non*-polygons, like Circles.  
     
   Copy the classes/interfaces from lesson5.lecture.intfaces2 into a new package for this Lab problem, and create a new Polygon interface. Then think of a way to make use of both ClosedCurve and Polygon so that, when computeAveragePerimeter is called on a ClosedCurve that implements the Polygon interface, the side lengths are added up, but when the object is not a polygon, a different computation of perimeter is done (as in the case of a circle). *Hint.* Create a default method in Polygon.  
     
   Try out your approach by adding two new ClosedCurves to your package: EquilateralTriangle and Ellipse (an equilateral triangle is a triangle in which all side lengths are equal). Modify DataMiner so that it includes in the objects list instances of these new classes.

*Hint.* The perimeter (or circumference) of an ellipse is *4aE* where *a* is the length of the semi-major axis and *E* is the value of the elliptic integral evaluated at the ellipse’s eccentricity. You do not need to know these technical concepts; just include *a* and *E* as instance variables in your class, of type double, and include them as arguments to the Ellipse constructor.   
  


1. The code for Lab7 Prob3 includes the following classes/interfaces: Cache, StaticStorage, along with a driver class Main and a Customer class. StaticStorage is intended to store data that becomes available during the execution of the application, and this data needs to be accessible throughout the application for a certain period of time. It is reasonable to make StaticStorage a singleton. Since StaticStorage is going to play the role of a *cache*, it is also natural for StaticStorage to inherit from Cache. For simplicity, we have only one method in Cache: timeout(). This tells how long items will be allowed to stay in the cache. For this problem, refactor Cache and StaticStorage so that
   1. StaticStorage is a singleton (by making it an enum)
   2. The method timeout() can be accessed by StaticStorage through “inheritance” (and the static keyword is removed)

Draw a class diagram of the classes in your updated package.

1. In the lesson7.lab4.prob4 package, there is a class called ForEachExample that specifies, in its main method, a list of Strings. Use the Java 8 forEach method within the main method to print out the list so that *all Strings are in upper case*. To do this, you will need to define your own implementation of the Consumer interface.
2. Rework the Duck Application of Lab 5, Problem 1 so that Flyable and Quackable interfaces *are* used after all, but now use Java 8 interfaces. Rewrite your code with this approach.