**Lab 8**

1. Short answer
   1. For each lambda expression below, name the parameters and the free variables.

i. Runnable r = ()   
 {

int[][] products = new int[s][t];  
**for** (**int** i = 0; i < s; i++) {  
 for(int j = i + 1; j < t; j++) {   
 products[i][j] = i \* j;

}   
}

}

There are no parameters.

Free variables: s, t.

ii.Comparator<String> comp = (s, t)   
 {  
 if(ignoreCase == true) {  
 return s.compareToIgnoreCase(t);

} else {

return s.compareTo(t);

}

}

Parameters: s, t

Free variables: ignoreCase

* 1. An example of a method reference is:

Math::random

Its corresponding functional interface is Supplier<Double>. Do the following:

* 1. Rewrite this method reference as a lambda expression

Supplier<Double> prob1B = Math::random;

* 1. Put this method expression in a main method in a Java class and use it to print a random number to the console

public static void main(String[] args) {

Supplier<Double> prob1B = Math::random;

System.out.println(prob1B.get());

}

* 1. Create an equivalent Java class in which the functional behavior of Math::random is expressed using an inner class (implementing Supplier); call this inner class from a main method and use it to output a random number to the console. The behavior should be the same as in part ii.

1. *Comparators.* 
   1. Look at the code in the package lesson8.lecture.comparator2. Suppose we sort using the sort method in the EmployeeInfo class together with the NameComparator. Look at the compare method in the NameComparator: If two Employee objects have the same name, what is the return value of compare? This tells us that these Employee objects should be *equal*, but is this always true? Give an example of two Employee objects having the same name but that should *not* be considered equal. Rewrite the compare method so that, if compare does return 0, the Employee objects are indeed equal. (This issue is known as *consistency with equals.*)
   2. Fix the compare method, as in part A, for the Comparator used in lesson8.lecture.comparator3
   3. Fix the compare method, as in part A, for the lambda expression used to compare Employee objects in lesson8.lecture.lambdaexamples.comparator3

3. Consider the following lambda expression. Can this expression be correctly typed as a BiFunction? (See lesson8.lecture.lambdaexamples.bifunction.) (Hint: Yes it can.)

(x,y) -> {

List<Double> list = **new** ArrayList<>();

list.add(Math.*pow*(x,y));

list.add(x \* y);

**return** list;

};  
  
Demonstrate you are right by doing the following: In the main method of a Java class, assign this lambda expression to an appropriate BiFunction and call the apply method with arguments (2.0, 3.0), and print the result to console.

5. Redo lesson7.labs.prob3 in two different ways:

a. Use a lambda expression instead of directly defining a Consumer

b. Use a method reference in place of your lambda expression in part a.

6. Finish the Examples exercise that was given in class (file: *Lambda and Method Reference Exercises*)