Lab10

1.

A.

i. Free variables are -> s, t, and Paramters are none.

Ii. Free variables are -> ignoreCase and Parameters are s, t

B.

```
class RandomGenerator {
  public static void main(String[] args) {
     Supplier<Double> d = Math::random;
     Supplier<Double> dMethod = () -> Math.random();
     Supplier<Double> dInner = new RandomGenerator.RandomGeneratorFunction();
  }

  static class RandomGeneratorFunction implements Supplier<Double> {
     @Override
     public Double get() {
        return Math.random();
     }
  }
}
```

2.

A.

Example of employees that would be considered equal when they shouldn't is Employees with the same name but different salaries. And the return value wouldn't be consistent. Same input could provide different results. So the compare method should use both name and salary properties.

```
public class EmployeeNameComparator implements Comparator<Employee> {
   @Override
   public int compare(Employee e1, Employee e2) {
      int nameComparison = e1.name.compareTo(e2.name);
      if (nameComparison != 0) {
         return nameComparison;
      }
      return Integer.compare(e1.salary, e2.salary);
   }
}
```

B.

```
public void sort(List<Employee> emps, final SortMethod method) {
    class EmployeeComparator implements Comparator<Employee> {
        @Override
        public int compare(Employee e1, Employee e2) {
            if (method == SortMethod.BYNAME) {
                int nameCompare = e1.name.compareTo(e2.name);
                if (nameCompare != 0) {
                    return nameCompare;
                }
                return Integer.compare(e1.salary, e2.salary);
            } else {
                int salaryComparison = Integer.compare(e1.salary, e2.salary);
                if (salaryComparison != 0) return salaryComparison;
                return e1.name.compareTo(e2.name);
            }
        }
    }
    Collections.sort(emps, new EmployeeComparator());
}
```

C.

```
public void sort(List<Employee> emps, SortMethod method) {
    Collections.sort(emps, (e1, e2) ->
    {
        if (method == SortMethod.BYNAME) {
            int nameComparison = e1.name.compareTo(e2.name);
            if (nameComparison != 0) return nameComparison;
            return Integer.compare(e1.salary, e2.salary);
        } else {
            int salaryComparison = Integer.compare(e1.salary, e2.salary);
            if (salaryComparison != 0) return salaryComparison;
            return e1.name.compareTo(e2.name);
        }
    });
}
```

3. Yes, That lambda expression can be correctly typed as a BiFunction<Double, Double, List<Double>>. That matches the structure of:
BiFunction<T, U, R> where T = Double, U = Double, R = List<Double> (return type).

//Inner Class Version (like lesson8.lecture.lambdaexamples.bifunction)

```
import java.util.function.BiFunction;
import java.util.*;

public class Main {
     public static void main(String[] args) {
     class MyBiFunction implements BiFunction<Double, Double,
List<Double>> {
          @Override
          public List<Double> apply(Double x, Double y) {
               List<Double> list = new ArrayList<>();
               list.add(Math.pow(x, y));
               list.add(x * y);
               return list;
          }
     }

     MyBiFunction f = new MyBiFunction();
     System.out.println(f.apply(2.0, 3.0));
     }
}
```

//Lambda Version

```
import java.util.function.BiFunction;
import java.util.*;

public class Main {
     public static void main(String[] args) {
     BiFunction<Double, Double, List<Double>> f = (x, y) -> {
          List<Double> list = new ArrayList<>();
          list.add(Math.pow(x, y));
          list.add(x * y);
          return list;
     };
     System.out.println(f.apply(2.0, 3.0));
     }
}
```