[1]Lab Assignment:JAVA TRAINING-FRESHER-nucleus 2018
[OOPs, UML, Threads, File, Collections, Custom Annotation and Reflection]

Q1. Create a class Book which contains the following attributes:
author, title, price
include at least the following methods:
two constructors,get and set methods for the attributes and toString( )

Q2. Create a class TextBook which is a subclass of Book. It has an extra attributes:
courseID
Include at least the following methods:
a constructor,
get and set methods for the attributes,
toString( )
a main method: read a book's information from console, create a TextBook instance and display the TextBook's information.

Q3. Create a class BookCollection which contains
the owner's name and ,an array of books that the owner has,toString( ) that outputs all the books in the BookCollection in a nice format.
a method hasBook(Book b) which checks if the book b is contained in the array (we consider two books the same if they have the same title and author).
a method sort() that sorts the books in the array by the lexicographical order of the book title, and author.

Create your own BookCollection and check if you own a particular book:"Effective Java" by Bloch. Sort the BookCollection and output the BookCollection.

Q4. This lab is based on lab 2. Add a field isbn to the Book class. The constructor and setIsbn method should check the ISBN number (assume that you only accept 10 digit or 13 digit ISBN number, note that '–' is allowed in an ISBN code.) and throws MalFormedISBNException if the format is not correct.
Modify BookCollection class to make sure all the books added to the collection have good ISBN number. Add a method showBookAt(int index) in BookCollection that throws indexOutofBoundException.
Create a BookCollection and allow the user to query the book in a loop with different input index. The program should not be terminated for the improper input.
Test the program with different input. Hand in the different input and output.

1 Java Training: Rajeev Gupta rgupta.mtech@gmail.com

Q5. Consider file data:

```
 97.59780253225763
 23.705044359023198
 72.97025259152822
 18.986484094410137
 77.56528079180427
 88.5456385076513
 59.09494795452861
 72.71304984780839
 80.0202893029642
 29.58427968260707
 74.66713563267237
 27.40345943374961
 15.990164966686493
 58.852582668688534
 45.58743329596889
 77.2227556103568
 53.49035808405568
 93.5583604428736
 35.09314691785803
 9.812059847790467
 51.438605600928376
 6.081908597641594
 2.604194278086147
 99.43752090812772
 20.355993598952395
```

Put data into a file named data.txt, Read from data.txt all the doubles (edited by a user) and display the biggest one.

Q6. This question is based on your lab 2: the Book class.
Add functionality so that the program can read book information from a text file where each book's information is stored in one line.
The price of each book is changed to half of the original price as it is a SALE period.
 To practise ObjectStream, you will store all the books that were just read in (with price changed to half) to an ObjectOutputStream (file name is BookOjects.dat).
 Read again, but this time from file BookOjects.dat. output the information of all the books in a nice format to the screen.
Create a program for a user to search books stored in the file BookOjects.dat by a given author or book title.

Q7. This program will accept a sentence from a console and store in a String.
Use java.util.StringTokenizer class object (instantiate a StringTokenizer class object with the inputString and a blank character as arguments) to find out how many words (or tokens separated by a blank) in the inputString. Set up a String array of that size.
Parse the tokens from the StringTokenizer and store them in the string array in a loop.
Sort the String array alphabetically. (Check out java.util.Arrays class for sort methods.)
Display the sorted String array one element per line on console.

Q8. Code a MerchandiseInventoryTest program which will accept 11 merchandises from a redirected input text file.
Copy and paste the following data to a text editor and save it as input.dat file.

```
ItemId Qty  price
-------------------------
KLM6666 22 49.50
GHI3330 30 20.00
WXX9000 11 15.00
JKL4440 40 35.00
YZZ9990 30 25.00
PQR6660 7  76.00
DEF2220 8  63.44
STU7770 15 55.00
ABC1110 10 50.50
TVV8880 18 43.50
MNO5550 50 12.00
```

Store each merchandise (one per input line) in an ArrayList object. (Merchandise class consist of itemCode, quantity, unitPrice)
Sort the merchandise elements in the ArrayList in an ascending order by names.
Display the sorted merchandise elements (one per line).
Sort the merchandises in a descending order by price.
Display the sorted merchandise elements again in the new order (one per line).
(Hint: Add getName(), getPrice() and toString() methods to the Merchandise class to facilitate two sorts and display.)
(Hint: To sort ArrayList, Queue, etc., use java.util.Collections.sort(Collection c, comparator com).
 Comparator is an interface with one abstract method compare(object1, object2).
Call upon the Collections.sort() to sort the ArrayList with the following:

```
Collections.sort( mList, new NameComparator() ); // sort by name
```

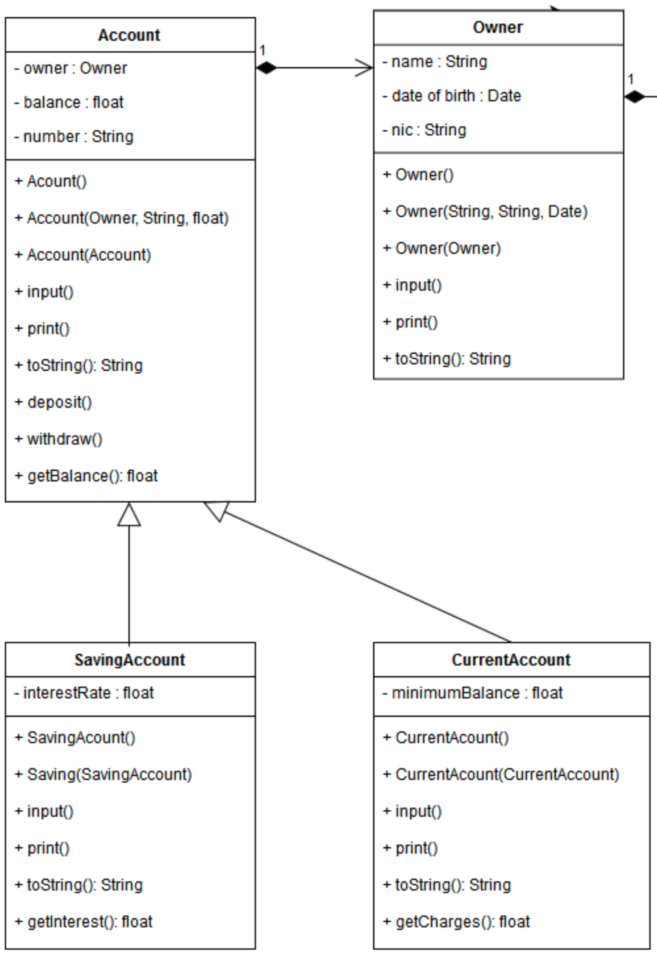
```
 Collections.sort( mList, new PriceComparator() ); // sort by price
```

Make sure the NameComparator and PriceComparater are defined as below:

```
  class NameComparator implements Comparator<Merchandise> {
    public int compare( Merchandise m1, Merchandise m2 ) {
           String s1 = m1.getName();
           return s1.compareTo( m2.getName() );
                                  // the String's method compareTo() returns -1, 0, 1
    }
  }
  class PriceComparator implements Comparator<Merchandise> {
    public int compare( Merchandise m1, Merchandise m2 ) {
            return (int)(m2.getPrice() -  m1.getPrice() );  // returns -1, 0, 1
    }
  }
```

)

Q9. Implement UML diagram



Q10. Create a database that holds the prerequisite relations of the computer science courses in our departments. To make it simple, you can just define a table with two fields: Course, PreCourse.

Course	**PreCourse**
cs383	cs255
cs483	cs254
cs483	cs383
cs487	cs255
cs375	cs255
cs255	cs162
cs254	cs162
cs162	cs161
cs365	cs255
cs465	cs365

Define the ODBC data source for the above database.

Create a Java program to manipulate the database that you just created.
print the prerequisite courses for course cs483
print all the courses that have cs255 as prerequisite.
change the cs375's prerequisite cs255 to cs365. (use update statement.)
insert a record for course cs256 (prerequisit cs255) to the table.
List all the course relationship, grouped by PreCourse. Also print the number of records in each group.

Q11. Demostrate deadlock with suitable example, explore approach to avoid deadlock from SO

<u>Lab assignment Custom Annotation and Reflection</u>

Create custom annotation to represent meta data as described:-Create annotation One is the Author Annotation and the other is the Version Annotation. This is a very common piece of meta-information that can be given to any Java file (or any other type of file).
Author.java

```
package reflections;

    import java.lang.annotation.*;
```

```
        @Target(value = {ElementType.CONSTRUCTOR, ElementType.METHOD,
ElementType.TYPE})
        @Retention(RetentionPolicy.RUNTIME)
        public @interface Author
        {
                String name() default "unknown";
        }
```
Note the definition of the Author annotation. This is a single-valued Annotation meaning that this Annotation has a single property called name. Also make a note of the Target Annotation. The presence of Target Annotation tells that Author Annotation can only be applied to Java elements like Constructor, Method and Type (Class/Interface). Another important thing to note is the Retention for this Annotation is set to Run-time because we want this Annotation information to be available to the JVM while the program is running. The name property is also given a default value unknown if the consuming Application fail to provide an explicit value.
Following is the definition of the Version Annotation. It looks exactly the same as Author Annotation except the fact that it has a property called number (which is of type double) to hold the version value.
Version.java

```
package reflections;

        import java.lang.annotation.ElementType;
        import java.lang.annotation.Retention;
        import java.lang.annotation.RetentionPolicy;
        import java.lang.annotation.Target;

        @Target(value = {ElementType.CONSTRUCTOR, ElementType.METHOD,
ElementType.TYPE})
        @Retention(RetentionPolicy.RUNTIME)
        public @interface Version
        {
                double number();
        }
```
Let us have a look at the following class definition that makes use of the above declared Annotations. The Annotations are applied at the class-level as well as in the method-level.
AnnotatedClass.java

```
package reflections;

        @Author(name = "Johny")
        @Version(number = 1.0)
        public class AnnotatedClass
        {
                @Author(name = "Author1")
                @Version(number = 2.0f)
                public void annotatedMethod1()
                {
                }

                @Author(name = "Author2")
                @Version(number = 4.0)
                public void annotatedMethod2()
                {
                }
        }
```

Now write an program to makes use of the new API to read the Annotation related information that are applied on various Java Elements. Write an utility method called readAnnotation() takes a parameter of type AnnotationElement which can represent a Class, method or Constructor. Then it queries for a list of Annotations of that particular element by calling the getAnnotations() method. Then the array is iterated to get the individual element, then made a downcast to extract the exact information – Author.name() and Version.number().