בס"ד

Part A - You should answer 5 / 6 of the following questions (10 points per question):

1. Write a method that given a Collection of doubles as input, returns a Collection with the following properties (= תכונות):
   A. Its content (= תוכן) is the content of the original Collection but <u>without</u> duplicates.
   B. If someone will iterate over it, the iteration order will be from smaller to bigger.

Sol :

```
public Collection<Double> answer(Collection<Double> doubles){
   return new TreeSet<>(doubles);
}
```

2. You are given a class Node:

```
public class Node implements Serializable {
   Node next;
   String name;

   public Node(Node next, String name){
       this.name = name;
       this.next = next;
   }

   private void setNext(Node next) {
       this.next = next;
   }

   private void setName(String name) {
       this.name = name
   }

   private void getName(String name) {
       return this.name;
   }

}
```

And we run the following code:

```
Node a = new Node(null, "a");
Node b = new Node(a, "b");
Node c = new Node(b, "c");
a.setNext(c);
```

A. What could be a problem with serializing the instance a? How does java handle it?

Sol: The problem could be that Java would get stuck in an infinite loop since c points back to a. The way Java handles it is by saving a copy of a and not serializing it again.

B. We add to the code above the following lines:

```
oos.writeObject(a); // oos is initialized to write to file "a.ser"
a.setName("b");
oos.writeObject(a);
//some code
a = (Node) ois.readObject(); // oos is initialized to read from file "a.ser"
System.out.println(a.getName());
a = (Node) ois.readObject();
System.out.println(a.getName());
```

What will be printed? Why?

Sol: “a” will be printed twice because java kept a copy of it as explained in the previous section.

3. A. Danny wrote the following enum. This breaks the Single Choice Principle! explain how.

```
public enum Operation {

   PLUS, MINUS;
   double eval(double x, double y) {
       switch (this) {
           case PLUS: return x + y;
           case MINUS: return x - y;
       }
       throw new UnsupportedOperationException();
   }
}
```

Sol: The problem is the list of operations (PLUS, MINUS) is saved twice, once on the declaration of the enum and another in the switch block.

B. Fix the enum such that the Single Choice Principle will be kept.

Sol: One way of solving it would be:

```
public enum Operation {

   PLUS{
       @Override
       double eval(double x, double y) {
           return x+y;
       }
   }
   , MINUS{
```

```
        @Override
        double eval(double x, double y) {
            return x-y;
        }
    };

     abstract double eval(double x, double y);
}
```

4. A. Explain open hashing and closed hashing.

Sol: In open hashing we allow items to map to the same cell in the table by keeping a linked list in each cell containing all elements that were mapped to that cell. Search is done accordingly. In closed hashing we do not allow collisions but instead for each insertion we look for the next open cell either linearly or quadratically, starting from the items hash.
Possible errors: adding duplicates, not instantiating the Collection.

B. Implement either (= או) closed or open hashing of Strings using any Collection. in the following Hashing class. To be specific : implement **contains** and complete (= השלם) **insert** methods. There is no delete method.
Assume that

- Resize is implemented properly and also remaps (= ממפה מחדש) all the items.
- You are given a proper upper load factor.
- You do not need to worry about updating the size or the capacity of the table.

You **cannot** assume that the collection you use is initialized.

```
public class Hashing {
  //code - define your Collection here

  public Hashing(double upperLoad){
      //initialize your collection here
  }

  public void insert(String s){
      double load = this.computeLoad();
      if(load > this.upperFactor){
          resize();
      }

      //code
  }

  public boolean contains(String s){
      //code
  }

  private void resize(){
```

```
      // dont implement
    }

    private double computeLoad(){
        // don't implement
    }
}
```

Sol: Open hashing:

```
public class Hashing {
    private int upperFactor;
    //code - define your Collection here
    LinkedList<LinkedList<String>> table;
    public Hashing(double upperLoad){
       table = new LinkedList<>();
    }

    public void insert(String s){
        //code
        double load = this.computeLoad();
        if(load > this.upperFactor){
            resize();
        }
        int location = s.hashCode() % table.size();
        if(table.get(location)==null){
            table.set(location,new LinkedList<>());
        }
        if(!table.get(location).contains(s))
            table.get(location).addLast(s);
    }

    public boolean contains(String s){
        //code
        int location = s.hashCode() % table.size();
        if(table.get(location)==null){
            return false
        }
        return table.get(location).contains(s);
    }

    private void resize(){
      // dont implement
    }

    private double computeLoad(){
        // dont implement
    }
}
```

Closed hashing:

```
public class Hashing {
   private int upperFactor;
   //code - define your Collection here
   LinkedList<String> table;
   public Hashing(double upperLoad){
      table = new LinkedList<>();
   }

   public void insert(String s){
       //code
       double load = this.computeLoad();
       if(load > this.upperFactor){
           resize();
       }
       int start = s.hashCode() % table.size();
       for(int i = 0; i<table.size(); i++){
           int location = (start + i) % table.size();
           if(s.equals(table.get(location))){
               return;
           }
           if(table.get(location)==null){
               table.add(location,s);
           }
       }
   }

   public boolean contains(String s){
       //code
       int start = s.hashCode() % table.size();
       for(int i = 0; i<table.size(); i++){
           int location = (start + i) % table.size();
           if(s.equals(table.get(location))){
               return true;
           }

       }
       return false;
   }

   private void resize(){
     // dont implement
   }

   private double computeLoad(){
       // dont implement
   }
}
```

5. A. You are given the following classes:

```
public class Outer<E> {

   public static class Inner{
       public E value;
       public Inner(E value){
           this.value = value;
       }
   }
}
```

What will be the result of running the code below? Why?

```
Outer.Inner i = new Outer.Inner(5);
System.out.println(i.value);
```

Sol: This will not compile since the generic parameter E is not recognized in the static inner class.

B. Now you are given the following:

```
public class SomeClass<T> {

   public <T> T someMethod(T val){
       return val;
   }
}
```

What will be the result of running the code below? Why?

```
SomeClass<String> c = new SomeClass<>();
int i = c.someMethod(5);
System.out.println(i);
```

Sol: This will print 5, since the generic parameter of the method is independent from the generic parameter of the class even though they are both named the same.

6. A. Given that Circle extends Drawable, what would be the result of the following?

```
Circle c = new Drawable();
System.out.print(c.toString());
```

Sol: This will not compile since Circle is a subclass of Drawable.

B.1. Explain upcasting and downcasting. Which one can be implicit? Which cannot? show an example of each.

Sol: Downcasting cannot be implicit but upcasting can. for example:

```
Circle c = (Drawable) d; \\ d is an instance of drawable
Drawable d = c; \\ c is a Circle
```

B.2. Will the following compile? explain why.

```
Object[] arr = new String[5];
```

Sol: . It will compile since arrays are covariant in Java.

Part B - Answer the following two questions (25 points per question) :

Question 1:

In this question you are given a huge inheritance tree, which describes the products in the Academon, starting with the root class AbstractProduct.

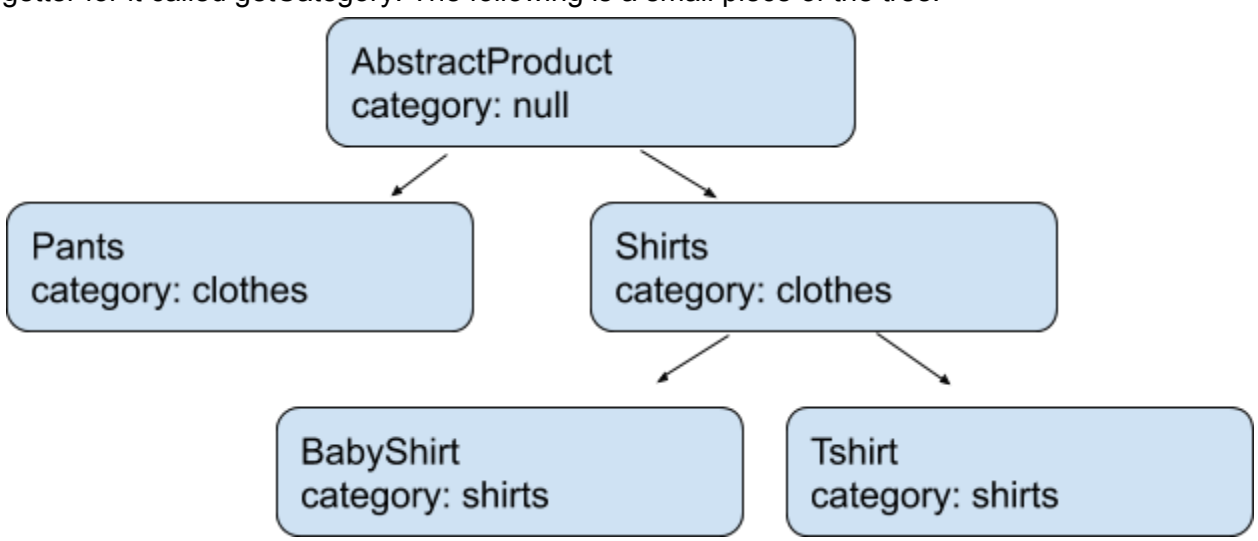
```
public abstract class AbstractProduct {
   protected int sellCount;
   protected String category = null;

   public boolean belongsToCategory(String cat){
       return false;
   }

   public int getSellCount(){
       return sellCount;
   }

   public String getCategory(){
       return category;
   }
}
```

Each inheriting class has a different category then its super class and an appropriate getter for it called getCategory. The following is a small piece of the tree.



A. I. Override the method belongsToCategory, such that for a given category returns true if the product belongs to that category or one of its parent classes belongs that category. For example the following will print true:

```
BabyShirt b = new BabyShirt();
System.out.println(b.belongsToCategory("clothes");
```

Sol:

```
protected boolean belongsToCategory(String cat){
   return this.getCategory().equals(cat) || super.belongsToCategory(cat);
}
```

II. Write the AbstractProduct method bestSellers:
Input: (1) a list of any products from the inheritance tree
(2) a category of type String.
Output: A new list with all products which belong to that category sorted by their sell count (use getSellCount).

Sol:

```
static LinkedList<? extends AbstractProduct> bestSellers(LinkedList<? extends
AbstractProduct> productList, String cat){
   LinkedList<AbstractProduct> res = new LinkedList<>();
   for(AbstractProduct a : productList){
       if(a.belongsToCategory(cat)){
           res.add(a);
       }
   }
   Collections.sort(res, (u,v) -> (u.getSellCount() - v.getSellCount()));
   return res;
}
```

B. In the following you may add new java files as you understand. A good answer is one that includes some code to show your intention (= כוונה).
I. Shirts and Pants may need to be washed (= לכבס). Washing takes time.
Given the inheritance tree above, you wish to have all classes inheriting **only** from Shirts and Pants, to have the capability (= יכולת) of being washed. Washing takes an int parameter for time.
Describe how you would implement this (you don't need to implement the washing itself)

Sol: Write an Interface:

```
public interface Washable{
   void wash(int time);
}
```

II. You notice there are many instances which you need to wash for exactly 1 hour time. How would you implement this default behavior?

Sol: Add the method:

```
public interface Washable{
   void wash(int time);
   default void oneHourWash(){
       wash(1);
   }
}
```

III. You now notice you need to count the amount of times an instance was washed. Describe how you would implement this without adding a counter to both Shirts and Pants classes.

Sol: Use composition. Create the class

```
public class Washer{
   int counter;
   Washer(Washable w){
       // save w
       counter = 0;
   }

   public void wash(int time){
       w.wash(time);
       counter++;
   }
}
```

Then instantiate one where necessary (you can use 'this' as a parameter). For example, in Pants class:

```
Washer washer = new Washer(this);
```

and delegate all washes to it:

```
public void wash(int time){
   washer.wash(time);
}
```

Question 2:

You are given the TennisPlayer class:

```
public class TennisPlayer {
   private int accuracy;

   public TennisPlayer(/* fields */){
       //code
   }
```

```
    public int hit(){
        //code
    }

    public TennisPlayer playAgainst(TennisPlayer tp){
       // dont implement
    }

}
```

A. You now want to add a special TennisPlayer which is capable (= מסוגל) of hitting the ball with topspin (סיבוב עליון =), and another special TennisPlayer who is capable of hitting the ball very hard. Sometimes you want to create a player that can both hit the ball with topspin and very hard.
Implement the above. You do not need to implement the special behaviors of the players in full details.

Sol: Using the decorator pattern. We define two classes:

```
public class HardHittingTennisPlayer extends TennisPlayer{
   TennisPlayer player;

   public HardHittingTennisPlayer(TennisPlayer player){
       this.player = player;
   }

   public int hit(){
       //decoration code
       player.hit()

   }

}

public class TopSpinTennisPlayer extends TennisPlayer{
   TennisPlayer player;

   public TopSpinTennisPlayer(TennisPlayer player){
       this.player = player;
   }

   public int hit(){
       //decoration code
       player.hit()

   }

}
```

This way we can make any combination of the two and create tennis players with other properties.

B. Note: The following is separate (= נפרד) from the previous section (= סעיף).
A TennisPlayer can play against another TennisPlayer by using the method playAgainst in TennisPlayer class.
In the implementation above, playAgainst receives another TennisPlayer to compete ((= להתחרות) against and returns the winner.
In the tennis world all tennis players are pros (= מקצוענים) or amateurs (= חובבנים).
We would like for games to be fair (= הוגן). Please make changes such that a pro player cannot compete with an amateur player. Trying to have an amatuer player play against a pro player should result in a **compile** time error.
You may change the TennisPlayer class and the playAgainst method in any way. You are not required to implement the method.

Sol: One option is to use marker (empty) interfaces:

```
public interface Amateur{

}

public interface Pro{

}
```

Now change the class declaration of the TennisPlayer to be generic:

```
public class TennisPlayer<T>
```

and the method playAgainst:

```
public TennisPlayer playAgainst(TennisPlayer<T> player){

}
```

This way you can instantiate tennis players as pros or amateurs such that having one play against the other is a compile time error.

C. The 'throwingMethod' is a method which can throw 3 types of checked Exceptions: TennisPlayException, DoubleFaultException and NetCallException.
DoubleFaultException and NetCallException both inherit from TennisPlayException.

I. Write the class TennisPlayException. Assume there are no other user defined Exceptions in the program.

Sol:

```
public class TennisPlayeException extends Exception{
```

```
}
```

II. Write the method signature (= חתימה) of ‘throwingMethod’ which receives no parameters and returns void.

Sol:

```
public void throwingMethod() throws TennisPlayeException{

}
```

III. Write a try-catch block that calls throwingMethod, and handles (= מטפל) errors in the following way: The DoubleFaultException and NetCallException are handled (= מטופלות) by throwing them up the stack, and the TennisPlayException is handled by first printing an error to the system (‘unrecognized exception’) and then throwing the exception up the stack.

Sol:

```
try{
   throwingMethod();
}catch(DoubleFaultException | NetCallException e){
   throw e;
}catch(TennisPlayException e){
   System.out.print("unrecognized exception");
   throw e;
}
```