בס"ד

יום ראשון ח' באלול התשע"ט
8.9.2019

האוניברסיטה העברית
בית הספר להנדסה ולמדעי המחשב ע"ש רחל וסלים בנין

מבוא לתכנות מונחה עצמים, סמסטר קיץ (67126)

מועד א'

מורה אחראית: הדס בר, מתרגל : יינון קפלן

ההוראות:

- משך הבחינה הוא 3 שעות.
- כל חומר עזר **אסור** בשימוש.
- למבחן שני חלקים: חלק א - 6 שאלות מתוכן עליכם לענות על 5 בלבד. (10 נקודות לכל שאלה)
  חלק ב - 2 שאלות, עליכם לענות על שתיהן. (25 נקודות לכל שאלה)
- עליכם לכתוב את התשובות **במחברת הבחינה** בלבד!
- אנא קראו כל סעיף **מתחילתו ועד סופו** לפני שאתם מתחילים לענות.
- אנא ציירו על הדף הראשון **במחברת הבחינה** את הטבלה הבאה:

	מספר השאלה	מספרי עמודים במחברת הבחינה בהם כתבתי את התשובה
חלק א		
חלק ב		

- המלצה חמה: "Less is more" - כתבו תשובות מדויקות שלא יותירו מקום לספק שאכן הבנתם.

בהצלחה רבה!!

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

1. Write a method which can accept (= לקבל) any List of type that **extends** Fruit, goes over them and filters out (= לסנן) all the rotten (= רקוב) fruits using the method isRotten. I.e your method returns the updated List.
   Here is a usage example of the method isRotten:

```
System.out.println(Boolean.toString(myLovelyPapaya.isRotten()));
```

Sol: Two possibilities -
With wildcard:

```
private List<? extends Fruit> filterRottenFruits(List<? extends Fruit>
fruits){
   Iterator<? extends Fruit> iter = fruits.iterator();
   while(iter.hasNext()){
       Fruit fruit = iter.next();
       if(fruit.isRotten()){
           iter.remove();
       }
   }
   return fruits;
}
```

Or with a generic method:

```
private <T extends Fruit> List<T> filterRottenFruits(List<T> fruits){
   Iterator<T> iter = fruits.iterator();
   while(iter.hasNext()){
       T fruit = iter.next();
       if(fruit.isRotten()){
           iter.remove();
       }
   }
   return fruits;
}
```

Note that it is a mistake to do the removal in a for each loop, as that could throw an exception:

```
for(Fruit fruit: fruits){
   fruits.remove(fruit); // wrong!!
}
```

2. A. Explain Erasure in java.

Sol: Erasure is the process which implements generics in Java. All generic types are casted to their compile time values. For example, given some List<String> list, and the following code:

```
String s = list.get(0);
```

The above compiles to:

```
String s = (String) list.get(0);
```

B. What will be the output of the following program?

```
Set<Number> set1 = new HashSet<>();
Set<Integer> set2 = new HashSet<>();
System.out.println(set2.getClass().equals(set1.getClass()));
```

Sol: Because of erasure both instances are of the same class, i.e. HashSet, and therefore ‘true’ will be printed.

C. A curious student added the following code to the program:

```
Set<Number> set = new HashSet<Integer>();
set.add(1.0);
```

What will be the result now? Why?

Sol: The code will not compile because generics are invariant, which means that if the generic type is different, even if one generic type inherits the other, the assignment of one to the other will fail.

3. A. Explain what is a Functional Interface in Java, and write such an interface explicitly (= באופן מפורש).

Sol: A functional interface is an interface with exactly one abstract method.

B. A functional interface can make your code cleaner. Can you show how? Explain by coding an example!

Sol: Given the following Interface

```
public Interface Example{
      public int doSomething(int i);
}
```

And the following method:

```
public void doesSomething(Example example){
      example.doSomething(2);
}
```

The following usage of doesSomething will compile:

```
doesSomething((int i) -> i*2);
```

4. Explain the following concepts and give an example:
   a. Facade design pattern
   b. Single Choice Principle
   c. Open Closed principle (no need for an example).

Sol: a. The Facade pattern is used to make complicated api's simpler by exposing a simple API which uses the complicated one. For example say we have an interface for sound manipulation, which has many different methods such as:

```
play(String filename, int volume, int bass, int treble);
```

A Facade for this could be the method:

```
play(String filename){
      // code
}
```

Which will call the more complicated method with some default values.

b. The Single Choice principle states that whenever a program needs to support some list of options, only one module should know the full extensive list. In our case a module could be some class.

For example say you have a program that receives a list of space ship types, then according to the Single Choice principle, only one class should know this full list, for example a factory which creates space ship instances.

c. The Open-Closed principle states that our software should be open for extension but closed to modification. That means that for example we can inherit some class to extend the software capabilities but we don't have to change the source code of that class.

5. A. Explain the Decorator pattern design in detail (= בפירוט).

Sol: The decorator pattern allows for extending object capabilities at runtime. In this pattern class A can inherit class B, and also receive an instance of B somehow (usually in the constructor). Upon calling class B method on an instance of A, A will decorate the call to B with some desired functionality, and delegate to B's instance where desired.
Reminder: delegate to B means calling a method of B instead of rewriting B's functionality.

B. Give an example, either (= או) of your design or in java, which uses this pattern.

Sol: A good example would be the InputStream class, which is decorated in java by the BufferedInputStream. Basically BufferedInputStream decorates InputStream and adds functionality that buffers large chunks of data instead of reading bytes one-by-one. Internally it uses InputStream methods.

C. Explain the benefits gained by using it and cons (= חסרונות) of implementing the example from (b) without the Decorator pattern.

Sol: Say we were to implement BufferedInputStream not as a decorator, but as a fully implemented class with the same functionality. Now say that we want to add the functionality of a ByteInputStream, in this case we need to create some class which implements both functionalities e.g. BufferedByteInputStream. This could explode exponentially since we have many desired functionalities. Having the decorator basically saves us all this work. The above could just be achieved by the call:

```java
InputStream stream = new ByteInputStream(new
BufferedInputStream(inputStream));
```

6. Given the class Node:

```java
public class Node {
   Node next;
   String data;

   public Node(String data){
       this.data = data;
       this.next = null;
   }

   public Node(String data, Node next){
       this(data);
       this.next = next;
   }
   public Node getNext() {
       return next;
   }

   public String getData() {
       return data;
   }
}
```

A. What could be the problem with the following iteration over the nodes?

```java
Node n3 = new Node("tail");
Node n2 = new Node("middle", n3);
Node n1 = new Node("head",n2);
Node node = n1;
try{
   while(true){
       node = node.getNext();
       foo(node.getData());
   }
}catch (NullPointerException e){

}
```

Sol: The problem could be that foo itself will throws a NullPointerException, and then the iteration would stop even though there are still more nodes.

B. Given 3 types of Exceptions: A, B and C.
For each Exception write the respective java file explicitly such that the following code will compile:
Restriction (= הגבלה): Your classes must inherit from the following classes BUT you may use each class as a parent only **once** :
RuntimeException, Exception, A, C. (As you can see B cannot be extended!)

```
public void foo(){
   try{
       throw new A();
   }catch (A a){

   }catch (C c){

   }catch (B b){

   }

   throw new B();
}
```

Sol: Class B must inherit from RuntimeException. Therefore A must inherit from C and C must inherit Exception (All other options should fail):

```
Class B extends RuntimeException{

}

Class A extends C{

}

Class C extends Exception{

}
```

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

Question 1:

"The most important thing in communication is hearing what isn't said."
(Peter Drucker)

In this question we will deal with communication (= תקשורת) between data centers.
Each data center includes (= מכיל):

1) Switches that communicate with each other.
2) Packets, which are the unit of data (= יחידת מידע) transmitted (= משודרת) from one switch to another.

Switches communicate with each other by forwarding packets.

IpPacket - which are a type of packet, have a source (= מקור) id (the id of the source switch), and a destination (= יעד) id (the id of the destination switch).
When a switch receives a packet he knows its destination according to the packet's destination id.
But Switches can also change the packet destination id using an operation (= פעולה) called **packet-encapsulation**.
The packet-encapsulation operation receives a new destination id - See code example next page.
A packet which has been encapsulated will always return the new id when its getDestinationId method is invoked (=נקראת/ מופעלת).
A packet can be encapsulated several times, i.e. its destination id can be changed more than once.
The opposite to packet-encapsulation is also possible! The opposite operation to encapsulation is called packet-deencapsulation.
packet-deencapsulation changes the destination id of the packet to its destination id before the encapsulation.
A packet that has been encapsulated *x* times can be de-encapsulated *x* times ; each deencapsulation will return the destination id of the packet one step backwards. See example below,
TcpPacket extend IpPacket and have a ttl (time-to-live) field that is decremented each time a packet crosses two switches.
Both encapsulation and de-encapsulation affect only the destination id, and not any other fields of the class.

In the following example, the tcpPacket instance starts with destination id 1:

```
try{

   System.out.println(tcpPacket.getDestinationId()); //1

   tcpPacket.encapsulate(2);
   System.out.println(tcpPacket.getDestinationId()); //2

   tcpPacket.encapsulate(3);
   System.out.println(tcpPacket.getDestinationId()); //3

   tcpPacket.deencapsulate();
   System.out.println(tcpPacket.getDestinationId()); //2

   tcpPacket.deencapsulate();
   System.out.println(tcpPacket.getDestinationId()); //1

   tcpPacket.deencapsulate(); // NotEncapsulatedException
}catch(NotEncapsulatedException e){

}
```

Notice in the last line the packet is not encapsulated so an appropriate exception is thrown.

In this question you are given the interfaces:

```
public interface Encapsulatable{
    IpHeader encapsulate(int newDestId);
    IpHeader deencapsulate() throws NotEncapsulatedException;

}

public interface IpHeader extends Encapsulatable {
    int getDestinationId();
    int getSourceId();
}
```

and the classes:

```
public class TcpPacket extends IpPacket{
   private IpHeader ipHeader;
   public int ttl;


   public TcpPacket(IpHeader  ipHeader, int ttl){
       super(ipHeader);
       this.ttl = ttl;
       this.ipHeader = ipHeader;
   }


   public int getDestinationId(){
       return ipHeader.getDestinationId();
   }

   public int getSourceId(){
       return ipHeader.getSourceId();
   }


   @Override
   public TcpPacket encapsulate(int destination) {
       this.ipHeader = this.ipHeader.encapsulate(destination);
       return this;
   }

   @Override
   public TcpPacket deencapsulate() throws
NotEncapsulatedException {
       this.ipHeader = this.ipHeader.deencapsulate();
       return this;
   }

}
```

```
public class IpPacket implements IpHeader {

   private int destinationId;
   private int sourceId;

   protected IpPacket(IpHeader packet){
       this.destinationId = packet.getDestinationId();
       this.sourceId = packet.getSourceId();
   }

   public IpPacket(int sourceId, int destinationId){
       this.sourceId = sourceId;
       this.destinationId = destinationId;
   }

   public int getDestinationId(){
       return this.destinationId;
   }

   public int getSourceId(){
       return this.sourceId;
   }

   @Override
   public IpPacket encapsulate(final int destination){
       //code

   }

   @Override
   public IpPacket deencapsulate() throws
NotEncapsulatedException{
       //code
   }
}
```

1. Please implement the encapsulation mechanism. Your implementation should support any number of encapsulations and de-encapsulations recursively. I.e any number of encapsulation and de-encapsulation operations of the same packet. For this section you **must** use some Collection from java.util.* and you may add or change any part of the given code, in any way **except** : (= חוץ מ) changing method signatures (=חתימות).
   A legal implementation is one that changes only the destination id, other fields **should not** be changed as a result of encapsulation and de-encapsulation.
   You do not have to copy the entire given code to your notebook, but be clear and explain where you added/changed.

Sol: Sol: Simply keep a LinkedList<Integer> as a protected member, and keep track of the destinationId's. This can all be done in IpPacket class. In the constructor initialize the list, then

```
@Override
public IpPacket encapsulate(final int destination){
   idList.addFirst(destination);
   return this;
}

@Override
public IpPacket deencapsulate() throws NotEncapsulatedException{
   if(idList.size() > 0){
       idList.pop();
       return this;
   }
   throw new NotEncapsulatedException();
}
```

And in TcpPacket:

```
public int getDestinationId(){
   return idList.peek();
}
```

2. Answer section (1) (= סעיף 1) again, this time you are allowed to add code **only** inside the IpPacket class methods encapsulate and de-encapsulate: No other code is allowed. Hint: inner classes!

Sol: One possible implementation

```
@Override
   public IpPacket encapsulate(final int destination){
       return new IpPacket(this){
           @Override
           public int getDestinationId(){
               return destination;
           }

           @Override
           public IpPacket deencapsulate() throws NotEncapsulatedException{
               return IpPacket.this;
           }
       };

   }

   @Override
   public IpPacket deencapsulate() throws NotEncapsulatedException{
       throw new NotEncapsulatedException();
   }
}
```

This is also possible with a local class which extends IpPacket.

3. Danny decided to implement the encapsulation mechanism in TcpPacket by returning an almost identical (= זהה) new instance of TcpPacket: He is doing it by using a copy-constructor, that will copy the object fields and change only the destination id to the new one. De-encapsulation is done by returning the original (= מקורי) packet, which he stored somewhere.
What could be the problem with this solution approach?

Sol: The problematic part is that the ttl field might change between encapsulation and de-encapsulation. The de-encapsulation will return a tcp packet instance with an invalid ttl field.

Question 2

“If you want to feel rich, just count the things you have that money can’t buy..”

In this question we will deal with the financial market!

1. Your first task is to create the Bank class. You need to create the Bank class in the singleton design pattern. In this section you do not need to add any other methods than the ones needed to support the Singleton pattern.

Sol: The Bank as a Singleton:

```
public final class Bank {
   private static Bank instance = null;

   private Bank(){
       //some code
   }
   public static Bank getInstance(){
       if(instance==null){
           instance = new Bank();
       }
       return instance;
   }

}
```

2. The Bank deals with a lot of money! Below is the Money class:

```
public class Money {
   private int value;
   private String serials;

   public Money(String serials, int value) {
       this.value = value;
       this.serials = serials;
   }

   public String getSerials(){
       return serials;
   }
```

```
    public int value(){
        return value;
    }

}
```

In the financial market, there are three periods:

1. VALUE period.
2. ALPHABETIC period.
3. ALPHABETIC_LENGTH period

The bank is responsible to sort its money. in each period the bank sorts its money in a different way.
In the VALUE period the bank sorts its money according to each money object value.
In the ALPHABETIC period the bank sorts the money according to alphabetical order of the serials.
In the ALPHABETIC_LENGTH period the bank sort the money according to the serials length, i.e. how many characters are in each serial.

Your task: write a public non-static method of the Bank called sort.
Receives:
(1) List of Money
(2) String - which can be only one of the above periods (See usage example bellow).
Return value : a sorted List according to the period given.
You may add helper functions in the Bank / Money classes and other helper classes.
Usage example:

```
// moneyList is of type List<Money>
bank.sort(moneyList, "ALPHABETIC") //sorts according to the
ALPHABETIC period
bank.sort(moneyList, "ALPHABETIC_LENGTH") //sorts according to
the ALPHABETIC_LENGTH period
```

Sol: Since we know only one of the three periods is valid, we can use an Enum:

```
public enum Period implements Comparator<Money> {
   VALUE((m1,m2) -> m1.value() - m2.value()),
   ALPHABETIC((m1,m2) -> m1.getSerials().compareTo(m2.getSerials())),
   ALPHABETIC_LENGTH((m1,m2) -> m1.getSerials().length() -
m2.getSerials().length());

   Comparator<Money> comparator;
   Period(Comparator<Money> comparator){
       this.comparator = comparator;
   }

   @Override
   public int compare(Money m1, Money m2){
       return comparator.compare(m1,m2);
   }
}
```

And in the bank class simply do:

```
List<Money> sort(List<Money> moneyList, String period){
   Collections.sort(moneyList, Period.valueOf(period));
   return moneyList;
}
```

3. Every new year, the bank prints more money.
   This year the bank decided to support not only int values, but fractions and very big numbers as well.
   A. Your first task is to rewrite Money class such that Money's value (i.e the value argument given to the constructor of Money class) can have any **type** that **extends** Number. The actual type should be determined at compilation time.

Sol: Money class that supports any type which extends Number as value:

```
public class Money<T extends Number> {
   private T value;
   private String serials;

   public Money(String serials, T value) {
       this.value = value;
       this.serials = serials;
   }

   public String getSerials(){
       return serials;
   }

   public T value(){
       return value;
   }

}
```

Inorder to print the money the bank uses the MoneyPrinter class which has a single method - print.
Given a money object the bank first needs to find the correct printer to use and then to call its print method. To achieve this, the bank class has a private field moneyMap of type HashMap<Money,MoneyPrinter> which it uses in the following way:

```
 //money is an instance of Money
moneyMap.get(money).print();
```

So, the Money class implementation should support this usage (= שימוש).
B. Your task is to make changes to the Money class so this would be possible.
Pay attention to methods and modifiers.
Note: Money objects are considered the same if their serials are the same.

Sol: A correct implementation is one that adds toString and hashCode to Money class. Notice that we also change the serials to be final.

```
public class Money<T extends Number> {
   private T value;
   private final String serials;

   public Money(String serials, T value) {
       this.value = value;
       this.serials = serials;
   }
```

```java
    public String getSerials(){
        return serials;
    }

    public T value(){
        return value;
    }

    @Override
    public boolean equals(Object other){
        Money otherMoney = (Money) other;
        return otherMoney.getSerials().equals(this.getSerials());
    }

    @Override
    public int hashCode(){
        return serials.hashCode();
    }

}
```

Good luck!!!