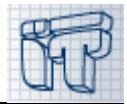


FIRST NAME:
LAST NAME:
NIA:
GROUP:

## First midterm exam

## Second Part: Problems (7 points out of 10)

Duration: 90 minutes
Highest score possible: 7 points
Date: March 9, 2018

Overall instructions for the exam:

- Books, notes, mobile phones, as well as other electronic devices are not allowed during the exam. Breaking this rule may result in expulsion from the examination
- Complete your personal information before starting the exam.

## Problem 1 (5 / 7 points)

The supermarket "CityMarket" wants to hire our company to develop an application that allows collecting orders and issuing the corresponding invoices for these orders. "CityMarket" sells two types of products, fresh products and dairies. In the fresh product category, "CityMarket" sells meat, vegetables and fruit. The team in charge of developing the communication interface with the end user's system sends us the following piece of code and its expected output on screen (our solution must be compatible with this piece of code).

```
Veggie v = new Veggie("Tomato", 1.10f, 3.1f);
Meat m = new Meat("Pork loin", 3.04f, 1.25f);
Fruit f = new Fruit("Apple", 4.50f, 1.50f);
Dairy d = new Dairy("Milk", 0.88f);
Product[] p = {v,m,f,d};
Order o = new Order();
o.add(p);
try{
   System.out.println(o.generateInvoice());
} catch (EmptyOrderException e){
   System.err.println("Empty order");
}
```

```
Invoice number 1
Tomato: 3.41€
Pork loin: 3.8€
Apple: 6.75€
Milk: 0.88€
TOTAL PRICE: 14.84€
```

### Section 1.1 (1 point)

Implement the class `Product`. This class represents each individual product sold by the supermarket. Each product has a name and a price, which are only visible from this class. In the example above, a unit of the product "Milk" costs 0.88€, while the product "Tomato" costs 1.10€ per kilogram. This class also declares a method `calculatePrice()`. Nevertheless, it is not possible to calculate the price of a generic product.

### Section 1.2 (0.5 points)

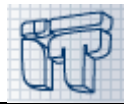
Implement the class `Dairy`. This class represents dairy products. In the case of dairy products, the method `calculatePrice()` simply returns the price of that dairy product.

### Section 1.3 (0.75 points)

Implement the class `Fresh`. This class represents fresh products (meat, vegetables, fruits). Each fresh product has a weight. In the case of fresh products, the method `calculatePrice()` returns the price of that product by its weight. NOTE: the subclasses of `Fresh` will be implemented by other team.

### Section 1.4 (2.75 points)

Implement the class `Order`. This class represents product orders. Each order has a unique identifier assigned in incremental order (the picture above shows the invoice for the first order), and a list of products. At the time of creating an order the number of products it will have is unknown, so we will use an object



of type `ArrayList<E>` to store the list of products. The class `Order` will have two methods, one to add an array of products to the object of type `ArrayList<E>`, and another one to generate an invoice (in `String` format), which looks like the one in the picture above for the example code provided. Be aware that the latter method will throw an `EmptyOrderException` (which has been already programmed by another team) if the order does not contain any product when trying to generate the invoice.

NOTE 1: In a String, the line return is programmed by concatenating `"\n"` to the String
NOTE 2: Do not care about importing the class `ArrayList<E>`.
NOTE 3: Some of the methods of `ArrayList<E>`, which may be useful for this section are:

- `boolean add(E e)`
- `void add(int index, E element)`
- `E get(int index)`
- `int indexOf(Object o)`
- `boolean isEmpty()`
- `E set(int index, E element)`
- `int size()`

## Problem 2 (2 / 7 points)

In the course "Electromagnetic Communications" at Universidad Carlos III de Madrid, students are asked to implement a Communication System which sends two signals at different frequencies depending on if the learner touches an apple, a banana, an apricot or does not touch anything. These two signals codify 2 bits of information using frequency modulation (FM). The following table represents how the information is codified:

| Bits | Frequency 1 | Frequency 2 | Information |
|---|---|---|---|
| 00 | 500 Hz | 500 Hz | Nothing has been touched |
| 01 | 500 Hz | 1000 Hz | The apple has been touched |
| 10 | 1000 Hz | 500 Hz | The banana has been touched |
| 11 | 1000 Hz | 1000 Hz | The apricot has been touched |

When both signals arrive to the receiver, the receiver can decode them and determine the fruit that was actually touched depending on the frequency values. However, as the signals can contain noise, it may happen that received frequencies are not the same as emitted frequencies. Because of that, an intermediate value must be taken (i.e., 750 Hz), to check if received frequencies are above or below that threshold, instead of assuming that received frequencies are exactly the same as the emitted frequencies.

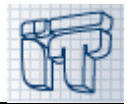
We want to implement a program to check the functionality of the decoder. However, we do not have the electronic circuits which send and receive the signals. Therefore, you are asked to design a Mock class which allows simulating the input signals.

### Section 2.1 (1 point)

Implement the class `ReceiverMock`. This class simulates the receiver which gets as input the two signals received (`freq1` and `freq2`) with their frequencies. These frequencies will be indicated when instantiating an object of the class `ReceiverMock`. The class contains a method `decode()`, which returns a String that can be "Nothing", "Apple", "Banana" or "Apricot", depending on the input frequencies.

### Section 2.2 (1 point)

Implement the class `ReceiverMockTest`. This class contains a method with JUnit (`testDecoder()`) that performs the necessary **white-box tests** to achieve 100% of method and line coverage. NOTE: Do not care about importing the classes you need to use JUnit.



# REFERENCE SOLUTIONS (several solutions are possible)

## PROBLEM 1

### Section 1.1 (1 point)

```
public abstract class Product {
    private String name;
    private float price;

    public Product(String name, float price){
        this.name = name;
        this.price = price;
    }

    public abstract float calculatePrice();

    public String getName(){
        return name;
    }

    public float getPrice(){
        return price;
    }
}
```

Evaluation criteria:

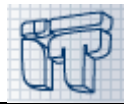
- 0.10: Class declaration
- 0.15: Attribute declaration as private
- 0.25: Constructor initializing the attributes
- 0.25: Abstract method `calculatePrice()`
- 0.25: Getters needed to retrieve the values of the attributes from other classes
- Significant errors are subject to additional penalties

### Section 1.2 (0.5 points)

```
public class Dairy extends Product {

    public Dairy(String name, float price){
        super(name, price);
    }

    public float calculatePrice() {
        return getPrice();
    }
}
```

Evaluation criteria

- 0.10: Class declaration extending from Product
- 0.15: Constructor calling the constructor of the superclass
- 0.25: Method `calculatePrice()`
- Significant errors are subject to additional penalties



## Section 1.3 (0.75 point)

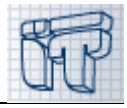
```
public class Fresh extends Product {
    private float weight;

    public Fresh (String name, float price, float weight) {
        super(name, price);
        this.weight = weight;
    }

    public float calculatePrice() {
        return getPrice()*weight;
    }
}
```

Evaluation criteria

- 0.10: Class declaration extending from Product
- 0.15: Attribute weight of type float (it can be private or protected)
- 0.25: Constructor calling the constructor of the superclass and initializing weight
- 0.25: Method `calculatePrice()`
- Significant errors are subject to additional penalties

## Section 1.4 (2.75 points)

```
public class Order {

    private int id;
    private static int counter = 1;
    private ArrayList<Product> items;

    public Order(){
        this.id = counter;
        this.items = new ArrayList<Product>();
        counter++;
    }

    public void add(Product[] products){
        for (int i = 0; i < products.length; i++){
            items.add(products[i]);
        }
    }

    public String generateInvoice() throws EmptyOrderException{
        if(items.size()==0){
            throw new EmptyOrderException("");
        } else{
            String s = "Invoice number " + id + "\n";
            float totalPrice = 0;
            for (int i = 0; i < items.size(); i++){
                float price = items.get(i).calculatePrice();
                s = s + items.get(i).getName() + ": " + price + "€ \n";
                totalPrice = totalPrice + price;
            }
            return s + "TOTAL PRICE: " + totalPrice + "€";
        }
    }
}
```



Evaluation criteria

- 0.25: Class declaration and attributes (including here the use of Product as type for the ArrayList)
  - If no static attribute is used: max 0.1
- 0.25: Constructor initializing correctly the ArrayList (no additional penalties for not using Product as type) and setting the id
- 0.75: Method `add`
  - Method signature: 0.25
  - Loop: 0.25
  - Addition of the elements to the ArrayList: 0.25
- 1.5: Method `generateInvoice()`
  - Method signature including `throws`: 0.25
  - Empty ArrayList case throwing the exception: 0.25
  - Loop: 0.25
  - Calculation of the price: 0.5
  - Construction and return of the String: 0.25 (minor error constructing the String are allowed)
- Significant errors are subject to additional penalties

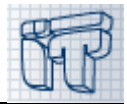
## PROBLEM 2

### Section 2.1 (1 point)

```
public class ReceiverMock {
	private int freq1;
	private int freq2;

	public ReceiverMock(int freq1, int freq2) {
		this.freq1 = freq1;
		this.freq2 = freq2;
	}

	public String decode(){
		if(freq1 < 750 && freq2 < 750) { // 00
			return "Nothing";
		} else if (freq1 < 750){ // 01
			return "Apple";
		} else if (freq2 < 750){ // 10
			return "Banana";
		} else { // 11
			return "Apricot";
		}
	}
}
```

### Section 2.2 (1 point)

```
public class ReceiverMockTest {

	@Test
	public void testDecoder(){
		assertEquals(new ReceiverMock(500,1000).decode(), "Apple");
		assertEquals(new ReceiverMock(1000,500).decode(), "Banana");
		assertEquals(new ReceiverMock(500,500).decode(), "Nothing");
		assertEquals(new ReceiverMock(1000,1000).decode(), "Apricot");
	}
}
```



## Section 2.1 (1 point): ReceiverMock

- 0.10: Attributes
- 0.15: Constructor
- 0.75: Method decode()
    - 0.15 Method declaration. If static is used or the method does not return a String, then 0.
    - 0.3 The conditions if/else to decode the frequencies are correct
    - 0.3 The method correctly returns the String. If there is no return for all cases, typically because there is an “if”, and several “else if”, but there is not an “else”, then 0.
- Significant errors are subject to additional penalties

## Section 2.2 (1 point): ReceiverMockTest

- 0.2: Class and method declaration with @Test. If @Test if not used, maximum 0.1.
- 0.8: Method testDecoder(): 0.2 for each case. The criteria for each case is the following.
    - If the method decode() is called in a static way, then 0 in that case.
    - If the String is not the expected output of the input given, maximum 0.05 in that case
    - If the method assertEquals() is not properly used, then 0.
- Significant errors are subject to additional penalties