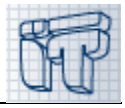


FIRST NAME:
LAST NAME:
NIA:
GROUP:

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

Duration: 180 minutes
Highest score possible: 7 points
Date: May 30, 2019

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.
- The exam must be filled in with blue or black pen. The final version of the exam must not be filled in with pencil.

## Problem 1 (2 points)

We want to develop an app for a sport event that allows printing the lists of volunteers with their identification number, the assigned position in the event (*"press"*, *"floaters"*, *"ticketing"*, *"grandstand"*, *"protocol"*), their shift (morning "*Shift: M*" or afternoon "*Shift: A*") and their personal data: age, gender and identification number (ID), as it is shown below:

```
Volunteer number: 1, Position: press, Shift: M, Age: 24, Gender: F, ID: 00000001-R
Volunteer number: 2, Position: grandstand, Shift: M, Age: 56, Gender: F, ID: 00000002-W
Volunteer number: 3, Position: ticketing, Shift: A, Age: 43, Gender: M, ID: 00000003-A
```

Moreover, volunteers receive a lunch ticket if they go to the event in the morning shift, and they receive a dinner ticket if they go to the event in the afternoon shift. Volunteers in the *ticketing* position (those who sell tickets) cannot receive dinner tickets because the ticket office closes at 18:00. For this app, you are given the following code:

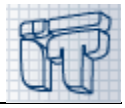
- The code of the class `Person`, which models the personal information of the volunteers (age, gender and id).
- Interface `Position`, which declares the method `selectPosition()`. This method returns a String with the assigned position of the volunteer.

```java
public class Person {
  private int age;
  private char gender;
  private String id;
  public Person(int age, char gender, String id) {
    this.age = age;
    this.gender = gender;
    this.id = id;
  }
  public String toString() {
    return "Age: " + age + ", Gender: " + gender +
      ", ID: " + id;
   }
}
```

```java
public interface Position {

String[] positions =
{"press", "floaters", "ticketing",
"grandstand", "protocol"};

String selectPosition();
}
```

### Section 1 (0.2 puntos)

Declare the interface `TicketPrinter` and its method `restaurantTickets()`. This method does not receive any argument, returns a String, and may throw the exception `TicketException`.



**Section 2 (1.5 puntos)**

Implement the class `Volunteer`, which inherits from the class `Person` and implements the interfaces `Position` and `TicketPrinter`. Moreover, this class contains the attributes and methods needed to model the state and behavior of a volunteer. In order to implement this class, you are asked to do the following things:

**(A)** Declare **4 attributes** that are not accessible from any other class, and **2 constants** that can be accessed from any other class. These attributes and constants are:

- `numTotal`: Integer number (`int`) whose initial value is 1 and is incremented each time a new volunteer is registered. This value is shared by all the created volunteers.
- `numVolunteer`: Integer number (`int`) whose value is assigned in a consecutive way for each volunteer so that each volunteer has a different value of `numVolunteer`.
- `position`: Attribute of type String used to store the assigned position.
- `shift`: Attribute of type char used to indicate the shift. It can take the value of the following constants.
    - *MORNING*: It takes the value 'M' and represents the morning shift.
    - *AFTERNOON*: It takes the value 'A' and represents the afternoon shift.

**(B)** Implement the **method** `selectPosition()` of the interface `Position`. This method assigns one of the five defined positions (*"press"*, *"floaters"*, *"ticketing"*, *"grandstand"*, *"protocol"*) to each volunteer in a random way. **NOTE**: In order to implement this method, you should make use of the `random` and `round` methods of class `Math`, whose description is as follows:

- `public static double random()  // Returns a double value between [0,1]`
- `public static long round(double a) // Returns the closest long to the argument, with ties rounding up`

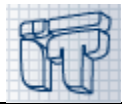
**(C)** Implement a **constructor** of the class `Volunteer` which receives as parameters the age, gender, id and shift, and assigns the rest of the attributes taking into account the following considerations:

- It must correctly initialize the attributes `numTotal` and `numVolunteer`.
- In order to assign the position of the volunteer, you need to call the method `selectPosition()` that you implemented previously.
- In order to initialize the attribute `shift`, you need to check inside the constructor whether the value of the shift is correct. You can call the method `boolean checkParameter(char shift)` of class `Volunteer`. If the value is correct, the attribute will be directly assigned. Otherwise, the attribute will be initialized with the default value (MORNING or 'M'). **NOTE**: You do not need to implement the method `checkParameter`. You can use directly and assume that it is correctly implemented and returns *true* if shift takes a valid value and *false* otherwise.

**(D)** Implement the **method** `toString()` which returns the information of the volunteer using the following format:

```
Volunteer number: <numVolunteer>, Position: <position>, Shift: <shift>, Age: <age>, Gender: <gender>, ID: <id>
```

**(E)** Implement the **method** `restaurantTickets()` of the interface `TicketPrinter`, which returns a String indicating the kind of restaurant ticket that each volunteer receives. For the case of volunteers with the morning shift, this method returns *"Lunch ticket"*, and for the case of the volunteers with the afternoon shift, it returns *"Dinner ticket"*. Furthermore, the method must throw an exception of type `TicketException` with the error message *"Invalid shift"* if the position is *ticketing* and the shift is afternoon (because ticket office closes at 18:00 and these volunteers do not get dinner tickets). **NOTE**: You do not need to implement the class `TicketException`. You can use it assuming that it is correctly implemented.



### Section 3 (0.3 points)

Complete the code of the method `main()` in class `PrintVolunteerList`. This method prints the list of volunteers and prints the corresponding restaurant tickets using the method `restaurantTickets()`.

```
public class PrintVolunteerList {

        public static void main(String[] args) {
                Volunteer v1 = new Volunteer(24, 'F', "00000001-R", Volunteer.MORNING);
                Volunteer v2 = new Volunteer(56, 'F', "00000002-W", Volunteer.AFTERNOON);
                Volunteer v3 = new Volunteer(43, 'M', "00000003-A", Volunteer.AFTERNOON);

                ArrayList<Volunteer> volunteers = new ArrayList<Volunteer>();
                volunteers.add(v1);
                volunteers.add(v2);
                volunteers.add(v3);

                // SECTION 3. COMPLETE
}
```

### Problem 2 (2 puntos)

You are given the classes `MyBasicLinkedList<E>` and `Node<E>`, which have the following already implemented methods:

```
public class MyBasicLinkedList<E> {
   private Node<E> first;
   public void setFirst(Node<E> first){...}
   public Node<E> getFirst(){...}
   public boolean isEmpty(){...}
   public void insert(E info){...}
   public E extract(){...}
   public int size(){...}
   public int numberOfOccurrences(E info){...}
   public MyBasicLinkedList<E> intersection
   (MyBasicLinkedList<E> list2){//SECTION 2}
}
```

```
public class Node<E> {
   private E info;
   private Node<E> next;
   public Node(E info){this.info = info;}
   public E getInfo(){...}
   public Node<E> getNext(){...}
   public void setInfo(E info){...}
   public void setNext(Node<E> next){...}
}
```

### Section 1 (0.25 points)

Program the class `MyBasicLinkedListException`, which inherits from class `Exception` and simply has a constructor which receives a message of type *String*.

### Section 2 (1.75 points)

Implement the method `public MyBasicLinkedList<E> intersection(MyBasicLinkedList<E> list2) throws MyBasicLinkedListException`

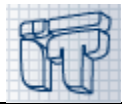
The signature of the method must be strictly followed in the solution of the exercise. This method receives and object of class `MyBasicLinkedList<E>` and returns a list whose elements are the result of the intersection between the two lists (i.e., the common elements). In the resulting list, it **does not matter** the order of the elements, but the list **cannot contain repeated elements**.

You are recommended to use the method `numberOfOccurrences` of class `MyBasicLinkedList` (this method returns the number of times `info` is in the list). You can assume that the method is correctly implemented. Moreover, the rest of the methods from both classes, except for `intersection`, are already correctly implemented and you can use them if needed.

As an example, if you have the following lists (L1 and L2) and the information is numeric:

`L1: 1 2 3 3`

`L2: 4 3 5 2 6 2 3 3 2 9`



The result of the method would be the following list: 2 3. The list with elements 3 2 would also be valid as the order of elements in the resulting list does not matter.

In addition, the method must throw the exception `MyBasicLinkedListException` if the list to be returned is empty. Furthermore, the original lists must preserve the elements they contain (in the same order) after the execution of the method.

### Problem 3 (2 points)

For this problem, you are given the interface `BTree<E>` and classes `LBNode<E>` and `LBTree<E>`, which model a binary tree. Moreover, you are given the class `BinaryTreeExample`, which has a `main` method and a method called `sumEvenNumbers`.

**NOTE:** There are not exceptions in the implementation given.

```
public interface BTree<E> {
  static final int LEFT = 0;
  static final int RIGHT = 1;

  boolean isEmpty();
  E getInfo();
  BTree<E> getLeft();
  BTree<E> getRight();
  void insert(BTree<E> tree, int side)
  BTree<E> extract(int side);

  String toStringPreOrder();
  String toStringInOrder();
  String toStringPostOrder();
  String toString();

  int size();
  int height();
  boolean equals(BTree<E> tree);
  boolean find(BTree<E> tree);
}
```

```
class LBNode<E> {
  private E info;
  private BTree<E> left;
  private BTree<E> right;

  LBNode(E info, BTree<E> left, BTree<E> right) {
    this.left = left;
    this.right = right;
    this.info = info;
   }

   E getInfo() {return info;}
   void setInfo(E info) {this.info = info;}

   BTree<E> getLeft() {  return left;     }
   void setLeft(BTree<E> left) {this.left = left;}

   BTree<E> getRight() {return right;}
   void setRight(BTree<E> right) {this.right = right;}
}
```

```
public class LBTree<E> implements
BTree<E>{
  private LBNode<E> root;
  public LBTree() {
    root = null;
  }
  public LBTree(E info) {
    root = new LBNode<E>(info,
           new LBTree<E>(),
           new LBTree<E>());
  }
  ...
}
```

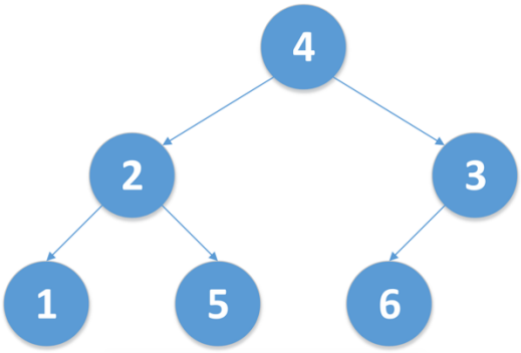
```
public class BinaryTreeExample {
   public static void main(String args[]) {
        BTree<Integer> n1 = new LBTree<Integer>(1);
        BTree<Integer> n2 = new LBTree<Integer>(2);
        BTree<Integer> n3 = new LBTree<Integer>(3);
        BTree<Integer> n4 = new LBTree<Integer>(4);
        BTree<Integer> n5 = new LBTree<Integer>(5);
        BTree<Integer> n6 = new LBTree<Integer>(6);

        // SECTION 1: Requested code must be written after
this comment.

        System.out.println("Sum even numbers = " +
                           sumEvenNumbers(n4));
    }
    public static int sumEvenNumbers(BTree<Integer> tree){
// SECTION 3  }
}
```

### Section 1 (0.5 points)

Complete the method `main` from class `BinaryTreeExample` to create the following binary tree (see next page), which stores information of type `Integer`, from the created nodes in the code given. You must use the methods from the abovementioned classes.

### Section 2 (0.3 points)

Given the binary tree provided in Section 1, indicate the sequences of pre-order, in-order and post-order traversals of the tree.




### Section 3 (1.2 points)

Implement the method `sumEvenNumbers` of class `BinaryTreeExample`, which sums the value of the nodes whose value is even. This method must be implemented in a **recursive way**. Any other implementation will not be graded.

**NOTE:** The sequence `System.out.println("Sum even numbers = " + sumEvenNumbers(n4));` must print the following result (when applied to the example tree):

```
Sum even numbers = 12
```

### Problem 4 (1 point)

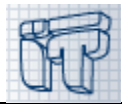
Given the following code:

```
import java.util.ArrayList;
public class BubbleSort {
public static void main(String[] args){
        ArrayList<Integer> a = new ArrayList<Integer>();
        a.add(2);
        a.add(5);
        a.add(4);
        a.add(6);
        a.add(8);
        a.add(3);
        a.add(1);
        a.add(9);
        a.add(7);
        System.out.println("Elements before sorting: ");
        System.out.println(a);
        System.out.println("Elements After sorting (in Descending order): ");
        bubbleSort(a);
        System.out.println(a);}
}
```

Implement the method `bubbleSort` (it does not return anything and it must be static) to sort the ArrayList in **descending** order.

Example:

```
Elements before sorting:
[2, 5, 4, 6, 8, 3, 1, 9, 7]
Elements After sorting (in Descending order):
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```



# REFERENCE SOLUTIONS (Several solutions may be valid for each of the problems)

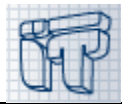
## PROBLEM 1

### Section 1 (0.2 points)

```
public interface TicketPrinter {
	String restaurantTickets() throws TicketException;
}
```

### Section 2 (1.5 points)

```
public class Volunteer extends Person implements Position, TicketPrinter {

	private static int numTotal;

	private int numVolunteer;
	private String position;
	private char shift;


	public static final char MORNING = 'M';
	public static final char AFTERNOON= 'A';


	public Volunteer(int age, char gender, String id, char shift) {
		super(age, gender, id);
		if (checkParameter(shift)){
			this.shift = shift;
		} else {
			this.shift = MORNING;
		}
		this.position = selectPosition();
		numTotal++;
		this.numVolunteer = numTotal;
	}

	public String toString() {
		return "Volunteer number: " + numVolunteer + ", Position: " + position +
", " + "Shift: " + shift + ", " + super.toString();
	}


	public String selectPosition() {
		int p = (int) Math.round(Math.random() * 4);
		return positions[p];
	}

	public String restaurantTickets() throws TicketException {
		if (shift == MORNING) {
		  return "Lunch ticket";
		}
		else if (shift == AFTERNOON && !this.position.equals("ticketing")) {
		  return "Dinner ticket";
		}
		else {
		  throw new TicketException("Invalid shift");
		}
	}
```



```
	public boolean checkParameter(char shift) { // For reference only
		if (shift == MORNING || shift == AFTERNOON) {
				return true;
		}
		return false;
	}
}
```

### Section 3 (0.3 points)

```
for(int i = 0; i<volunteers.size(); i++) {
	System.out.println(volunteers.get(i));
	System.out.println(volunteers.get(i).restaurantTickets());
}
```

### Section 1 (0.2 points)

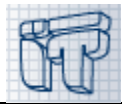
- 0.2: Declaration of the interface and abstract method
  - If the student does not show knowledge about interfaces because he/she writes abstract in the declaration, implements the method, or writes {} instead of ;, then 0
  - Do not penalize if public is indicated in the method regardless it is not needed (all methods are public in an interface)
  - Penalize 0.05 if throws is not used in the method
- Significant errors are subject to additional penalties

### Section 2 (1.5 puntos)

- 0.1: Class declaration
- 0.05: Declaration of the static variable numTotal
- 0.1: Declaration of variables numVolunteer, position y shift
- 0.1: Declaration of the two constants
- 0.5: Constructor
  - 0.1: Declaration
  - 0.1: Management and call to super()
  - 0.1: Management and assignation of variable shift
  - 0.1: Management and assignation of variable position
  - 0.05: Management and assignation of the static attribute
  - 0.05: Management and assignation of attribute numVolunteer
- 0.15: Method toString()
  - Do not penalize if the method is implemented with more line codes than needed
- 0.20: Method selectPosition()
  - 0.05: Declaration of the method
  - 0.10: Obtain position using Math.random
  - 0.05: Return the position
- 0.30: Method restaurantTickets()
  - 0.05: Declaration
  - 0.25: Conditions to print lunch (0.05) or dinner tickets (0.1), or throw the exception (0.1)
- Significant errors are subject to additional penalties

### Section 3 (0.3 points)

- 0.1: Traverse the ArrayList with the correct limits in the for loop.
- 0.1: Print the list of volunteers
- 0.1: Print the restaurant tickets
- Significant errors are subject to additional penalties



## PROBLEM 2 (2 points)

### Section 1 (0.25 points)

```
public class MyBasicLinkedListException extends Exception {
     public MyBasicLinkedListException(String msg){
           super(msg);
     }
}
```

### Section 2 (1.75 points)

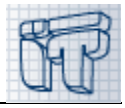
```
public MyBasicLinkedList<E> intersection(MyBasicLinkedList<E> list2) throws
MyBasicLinkedListException{
    MyBasicLinkedList<E> result = new MyBasicLinkedList<E>();
    Node<E> aux = this.getFirst();

    // Also valid for(int i=0; i<this.size(); i++){
    while (aux != null) {
        if ((list2.numberOfOccurrences(aux.getInfo()) != 0) &&
          (result.numberOfOccurrences(aux.getInfo()) == 0))
            result.insert(aux.getInfo());
        aux = aux.getNext();
    }

    // Also valid if (result.size()==0)
    if (result.isEmpty())
        throw new MyBasicLinkedListException("Empty intersection!");
    return result;
}
```

### Section 1 (0.25 puntos)

- 0 if the solution does not make sense and/or the solution is wrong in general
- 0.1: Correctly declaration of the class, extending Exception
- 0.15: Correct implementation of the constructor
- Significant errors are subject to additional penalties

### Section 2 (1.75 points)

- 0 if the solution does not make sense and/or the solution is wrong in general
- 0.1: Declaration and initialization of the resulting list
- 0.1: Correct access to the first element to traverse the list
- 0.25: Correct declaration of the loop (using as many iterations as the number of elements of one of the lists and defining a correct stop condition)
- 0.5: Correctly checking if the current element is in the other list and it is not repeated in the resulting list (0.25 each condition)
- 0.25: Correctly insertion of the element in the resulting list (in case the insertion is necessary)
- 0.15: Correct advancement in the list with the next element
- 0.25: Correctly throwing the exception if the resulting list is empty
- 0.15: Correctly returning the resulting list
- Penalize 0.2 if any of the lists is modified
- Significant errors are subject to additional penalties



## PROBLEM 3

### Section 1 (0.5 points)

```
n2.insert(n1, BTree.LEFT);
n2.insert(n5, BTree.RIGHT);
n3.insert(n6, BTree.LEFT);
n4.insert(n2, BTree.LEFT);
n4.insert(n3, BTree.RIGHT);
```

### Section 2 (0.3 puntos)

```
Pre-order = 4 2 1 5 3 6
In-order = 1 2 5 4 6 3
Post-order = 1 5 2 6 3 4
```

### Section 3 (1.2 points)

```
public static int sumEvenNumbers(BTree<Integer> tree) {
	if (tree.isEmpty()) {
		return 0;
	} else if (tree.getInfo() % 2 == 0) {
		return sumEvenNumbers(tree.getLeft()) +
			sumEvenNumbers(tree.getRight()) + tree.getInfo();
	} else {
		return sumEvenNumbers(tree.getLeft()) +
sumEvenNumbers(tree.getRight());
	}
}
```

### Section 1 (0.5 points)

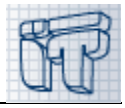
- 0.5: Inserts are correct regardless the second parameter of insert (0.1 each insert)
- Penalize 0.2 if the second parameter of the insert method is incorrect. Second argument may only take the following values (BTree.LEFT or BTree.RIGHT) or (0 or 1)
- Significant errors are subject to additional penalties

### Section 2 (0.3 points)

- 0.1 each traversal
- Significant errors are subject to additional penalties

### Section 3 (1.2 points)

- 0.3: Check whether the tree is empty or not
- 0.2: Check whether the information of the node is even or not
- 0.4: First recursive case when the information is even
- 0.3: Second recursive case (else) when the information is odd
- If the method is not implemented in a recursive way, then 0
- Significant errors are subject to additional penalties



## PROBLEM 4

```
public static void  bubbleSort (ArrayList<Integer> a) {
      for (int i = 0; i < a.size(); i++) {
            for (int j = 0; j < a.size() - i - 1; j++) {
                  if (a.get(j).comparteTo(a.get(j + 1)) < 0) {
                      Integer temp = a.get(j);
                      a.set(j, a.get(j + 1));
                      a.set(j + 1, temp);
                  }
            }
      }
}
```

## PROBLEM 4

- 0.1: Correct declaration of the method
  - Penalize 0.1 if the method is not declared as void and/or static, if the argument is not provided, and if the type ArrayList is incorrect
- 0.2: First for loop
  - Penalize 0.1 if students write length instead of size()
  - If limits are not correctly defined, then 0
- 0.2: Second for loop
  - Penalize 0.1 if students write length instead of size()
  - If limits are not correctly defined, then 0
- 0.3. Conditional if
  - Penalize 0.3 if sorting is carried out in ascending order
  - Penalize 0.2 if indexes of get methods are incorrect
- 0.2: Lines inside the if
  - Penalize 0.1 if the use of get is incorrect and/or the indexes used in the gets are incorrect
  - Penalize 0.1 if the use of set is incorrect and/or the indexes used in the sets are incorrect