FIRST NAME:
LAST NAME:
NIA:
GROUP:

Second Part: Problems (7 points out of 10)

Duration: 180 minutes
Highest score possible: 7 points
Date: June 28, 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.
- 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)

Section 1 (0.5 points)

A new cable television service has been developed in a telecommunications company. As a member of the company, you are in charge of designing some Java classes to model this service. First, you want to model television programmes using the `Programme` class. A programme has a `name`, a string of text to indicate the date and time it is broadcast (`time`) and the `quality` at which it is broadcast. All attributes can be visible in any class within the package but cannot be accessed from classes which belong to other packages. With respect to quality, this will be modeled with constants, which can take two values:

- `SD = 1`   `HD = 2`

This class will have a constructor that receives by parameter the initial values of the attributes. If the parameter that refers to quality does not meet the abovementioned values, the SD quality will be assigned automatically. In addition, the class has a `void description()` method to indicate the description of the programme, although in this class there is not enough information available to know what the code in this method should look like, since the description will depend on the type of programme. The platform defines four types of programmes: `News`, `Sport`, `Film` and `Series`. However, the code of the programme types will be programmed by another development team. In this section, you are asked **to write only the code** of the `Programme` class with the given specifications.

Section 2 (0.7 points)

On the other hand, you are in charge of client management. Initially two functionalities will be included, as shown in the following interface, **to be implemented by all types of clients**:

```
public interface Functions {
      void record(String name) throws TVException;
      void generateIncidence(String incidence);
}
```

Since not all clients can subscribe to television, two types of clients are defined: `Client` and `ClientTV`. The former will not be subscribed to television (s/he can have a landline, cell number, etc., but not TV). This client will be identified by an identifier (`id`) that will be provided in an incremental way (the first client will have id 1, the second id 2 and so on), a name (`name`), which will be a string and, in addition, an `ArrayList` of strings to store its incidences. All attributes will only be visible within the class. In this section you are asked to write the code of the class `Client` in which, apart from declaring the attributes, you must include at least:

- A constructor in which the attributes are initialized. This constructor only receives the client's name by parameter.

- The method `record(String name)`, which in this case will directly throw a `TVException` (you can assume this class is already coded) with the message "`Recordings are not available in your subscription`".
- The method `generateIncidence(String incidence)`, which adds a new incidence to the `ArrayList` of incidences.

Section 3 (0.8 points)

Finally, you are asked to code the class `ClientTV`, which models the clients who have subscribed to television. These clients, in addition to having a name, id and list of incidences, have also an array with the capacity to record up to 100 programmes (`Programme`), whatever their type. The empty positions of this array will be `null`. In order to record the programs, and as will be discussed later, you will need to make use of the `TVGuide` class, which is a class created by another team. The `TVGuide` class has a method `static Programme searchProgramme(String name)`, which returns a TV programme given its name if it is found in the guide or `null` if it is not found.

Moreover, the class `ClientTV` will include:

- A constructor that initializes all attributes. In addition, a promotional recording must be included within the recordings. It will be of type `News` with the name "`TV info`", time "`28/06/2018 10:30`" and available in HD. You can assume that the class that implements the news does not need any additional parameters with respect to generic programmes.
- The method `record(String programme)`, which will serve to record a programme. To do this, it will search for the program in the guide. If the programme does not exist, a `TVException` will be thrown indicating "`Programme not found`". If the program is found in the guide, an attempt will be made to insert it into the first available slot in the array of programmes to be recorded (you can assume that the programme has never been recorded, and therefore it will not appear twice in the array). If the array is full with 100 recordings (no `null` value position), a `TVException` would be thrown with the message ""`There is not space for more recordings`". Finally, if the operation is successful, a message will be shown on screen with the following format

  `Well done, <name>. Your programme has been scheduled for recording successfully.`
- The method `generateIncidence(String incidence)`, does not change its behavior with respect to customers who do not subscribe to television.

NOTE 1: you do not need to worry about importing the class `ArrayList<E>` in the first line of your classes.
NOTE 2: The class `ArrayList<E>` has the following methods, some of which can be useful in this problem:

- `boolean add(E e)`
- `void add(int index, E element)`
- `E get(int index)`
- `boolean isEmpty()`

Problem 2 (1 point)

The class `ExamUtil` has the following method;

```
    public static String gradeAlpha(float grade) {
        if (grade < 0.0f || grade > 10.0f)
            throw new IllegalArgumentException("Illegal grade");
        if (grade < 5.0f)            return "Failed";
        else if (grade < 7.0f)       return "Passed";
        else if (grade < 9.0f)       return "Outstanding";
        else                         return "Remarkable";
    }
```

NOTE: The grades are assumed to have **only one valid decimal place**. That is to say, grades with more than one decimal place are not to be considered.

Section 1 (0.1 point)

You are asked to write a test which tests the behavior of this method with the highest invalid negative grade.

Section 2 (0.1 point)

You are asked to write a test which tests the behavior of this method with the lowest invalid positive grade.

Section 3 (0.8 point)

You are asked to write a test **for each possible result** to check both the minimum and the maximum grade for that result. That is to say, the lowest and highest "failed" grade, the lowest and highest "passed" grade, the lowest and highest "outstanding" grade, and the lowest and highest "remarkable" grade.

Problem 3 (2 points)

Given a class `Node` with the following methods, and a class which implements a linked list (`LinkedList`) formed by a set of nodes:

Section 1 (1.25 points)

Write the method `replaceZerosByPrevSum()` included in the class `LinkedList`. This method modifies the current list, obtaining a list made up of the sums of the elements prior to each "0". Take a look at the examples below to better understand how the method works.

```
Example 1. List ending in "0":
In: 5 -> 5 -> 5 -> 0 -> 17 -> 17 -> 0 -> 4 -> 4 -> 0 -> null
Out: 15 -> 34 -> 8 -> null

Example 2. List not ending in "0":
In: 5 -> 5 -> 5 -> 0 -> 17 -> 17 -> 0 -> 4 -> 4 -> null
Out: 15 -> 34 -> 4 -> 4 -> null

Example 3. List starting with "0":
In: 0- > 5 -> 5 -> 5 -> 0 -> 17 -> 17 -> 0 -> 4 -> 4 -> null
Out: 15 -> 34 -> 4 -> 4 -> null
```

<u>**NOTE 1**</u>. The input list ensures that no two consecutive items with value "0" will appear.
<u>**NOTE 2**</u>. You may reuse the nodes of the original list to calculate the sums. You do not need to create new nodes or auxiliary lists.

```
public class Node {
    private Node next;
    private int num;

    public Node(int val, Node next){
        this.num = val;
        this.next = next;
    }
    public Node(int val) {
        this(val, null);
    }
    public int getInfo() {
        return this.num;
    }
    public Node getNext() {
        return this.next;
    }
    public void setInfo(int info) {
        this.num = info;
    }
    public void setNext(Node next) {
        this.next = next;
    }
 }
```

```
public class LinkedList {
    Node head;
    public LinkedList(int val) {
        this.head = new Node(val);
    }
    public LinkedList() { this.head = null;}
    public void insert(int val) {
       Node newNode = new Node(val);
       newNode.setNext(this.head);
       head = newNode;
    }
    public void print() {
        Node tmpNode = head;
        while (tmpNode != null) {
            System.out.print(tmpNode.getInfo()
                             + " -> ");
            tmpNode = tmpNode.getNext();
        }
        System.out.print("null");
    }
    public void replaceZerosByPrevSum() {
        // SECTION 1
    }
}
```

Section 2 (0.75 points)

Write a `Main` class (only with a `main` method) to run and display on screen the following example:

```
In: 5 -> 5 -> 5 -> 0 -> 17 -> 17 -> 0 -> 4 -> 4 -> 0 -> null
Out: 15 -> 34 -> 8 -> null
```

Problem 4 (2 points)

For the following problem, consider the `BTree<E>` interface, and the `LBNode<E>` and `LBTree<E>` classes with the following methods already implemented, with the same semantics as those we saw in the laboratory sessions of this course, and without worrying about exceptions.

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

```
public class LBNode<E> {
    private E info;
    private BTree<E> left;
    private BTree<E> right;
    LBNode(E info, BTree<E> left, BTree<E> right) { ... }
    E getInfo() { ... }
    void setInfo(E info) { ... }
    BTree<E> getLeft() { ... }
    void setLeft(BTree<E> left) { ... }
    BTree<E> getRight() { ... }
    void setRight(BTree<E> right) { ... }
}
```

```
public class LBTree<E> implements BTree<E> {
    private LBNode<E> root;
    public LBTree() { ... }
    public LBTree(E info) { ... }
    // ...
}
```

You are asked to add a new method `public boolean someSubtreeEqualsToSomeSubtreeOf(BTree<E> other)` to the class `LBTree<E>`, which given `this` tree, and another tree (`other`) as parameter, returns:

- `true` when it is possible to find some `other` non-empty subtree equal to some non-empty subtree of `this`
- `false` otherwise

The search should start with the largest subtrees and continue with the smallest subtrees. To solve it, it is enough to use the existing methods provided (including this new method if it is solved recursively), although you can add new methods to the classes and/or the interface if you consider it appropriate. The algorithm should stop when it finds the first match.

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

PROBLEM 1

Section 1 (0.5 points)

```
public abstract class Programme {
	static final int SD = 1;
	static final int HD = 2;

	String name;
	String time;
	int quality;

	public Programme(String name, String time, int quality){
		this.name = name;
		this.time = time;
		if(quality != SD && quality != HD)
			quality = SD;
		this.quality = quality;
	}

	public abstract void description();
}
```

Section 2 (0.7 points)

```
public class Client implements Functions {
	private static int id_counter;
	private int id;
	private String name;
	private ArrayList<String> incidences;

	public Client(String name){
		this.name = name;
		id_counter++;
		this.id = id_counter;
		incidences = new ArrayList<String>();
	}

	public void record(String name) throws TVException{
		throw new TVException ("Recordings are not available in your
subscription");
	}

	public void generateIncidence(String incidence){
		incidences.add(incidence);
	}

	public String getName() {
		return name;
	}
}
```

Section 3 (0.8 points)

```
public class ClientTV extends Client{
	private Programme[] recordings;

	public ClientTV(String name){
		super(name);
		recordings = new Programme[100];
		recordings[0] = new News("TV Info", "28/06/2018 10:30", Programme.HD);
	}

	public void record(String name) throws TVException{
		Programme p = TVGuide.searchProgramme(name);
		if(p == null){
			throw new TVException("Programme not found");
		}
		boolean inserted = false;
		for(int i=0; !inserted && i<recordings.length;i++){
			if(recordings[i] == null){
				recordings[i] = p;
				inserted = true;
			}
		}
		if(!inserted){
			throw new TVException("There is not space for more recordings");
		}
		else{
			System.out.println("Well done, " + getName() + ". Your programme
has been scheduled for recording successfully");
		}
	}
}
```

PROBLEM 1

Section 1 (0.5 points)

- 0.1: Class declaration
- 0.1: Constants declaration
- 0.1: Attributes. If visibility is not correct, then 0
- 0.1: Constructor
- 0.1: Abstract method
- Significant errors are subject to penalties

Section 2 (0.7 points)

- 0.1: Class declaration implementing the interface
- 0.1: Management of the id with the two attributes (one of them static) in the attributes part. If the static attribute is also declared as constant (final), then 0
- 0.1: Name and ArrayList of incidences declaration
- 0.1: Constructor
- 0.1: Method record()
- 0.1: Method generateIncidence()
- 0.1: Method getName()
- Significant errors are subject to penalties

Section 3 (0.8 points)

- 0.1: Class declaration. If not inheriting from Client or implementing the interface, then 0.
- 0.1: Attribute recordings.
- 0.1: Constructor
- 0.5: Method record()
    - 0.1: Search for the program in the guide. If there is a call from a TVGuide object, instead of from TVGuide.searchProgramme (as it is a static method), then 0
    - 0.2: Insertion in the array at the first available position
    - 0.1: Throwing exceptions in both cases as indicated. If error messages are printed instead of throwing exceptions, then 0
    - 0.1: Display the message if the operation is successful. If the name is accessed directly without getName(), then 0
- With respect to the generateIncidence() method, it will be valid if it is implemented with super.generateIncidence(incidence) or if it is not overwritten. Any other solution will be penalized with 0.1.
- Significant errors are subject to penalties

PROBLEM 2

Section 1 (0.1 points)

```
@Test(expected = IllegalArgumentException.class)
public void testUnderGrade() {
        ExamUtil.gradeAlpha(-0.1f);
}
```

Section 2 (0.1 points)

```
@Test(expected = IllegalArgumentException.class)
public void testOverGrade() {
        ExamUtil.gradeAlpha(10.1f);
}
```

Section 3 (0.8 points)

```
@Test
public void testFailedLow() {
        assertEquals(ExamUtil.gradeAlpha(0.0f), "Failed");
}
@Test
public void testFailedHigh() {
        assertEquals(ExamUtil.gradeAlpha(4.9f), "Failed");
}
@Test
public void testPassedLow() {
        assertEquals(ExamUtil.gradeAlpha(5.0f), "Passed");
}
@Test
public void testPassedHigh() {
        assertEquals(ExamUtil.gradeAlpha(6.9f), "Passed");
}
@Test
public void testOutstandingLow() {
        assertEquals(ExamUtil.gradeAlpha(7.0f), "Outstanding");
}
@Test
public void testOutstandingHigh() {
```

```
                assertEquals(ExamUtil.gradeAlpha(8.9f), "Outstanding");
        }
        @Test
        public void testRemarkableLow() {
                assertEquals(ExamUtil.gradeAlpha(9.0f), "Remarkable");
        }
        @Test
        public void testRemarkableHigh() {
                assertEquals(ExamUtil.gradeAlpha(10.0f), "Remarkable");
        }
```

Section 1 (0.1 points)

- 0.05: Calling gradeAlpha properly with -0.1
- 0.05: Checking the expected exception
- Significant errors are subject to penalties

Section 2 (0.1 points)

- 0.05: Calling gradeAlpha properly with 10.1
- 0.05: Checking the expected exception
- Significant errors are subject to penalties

Section 3 (0.8 points)

- 0.30: Calling gradeAlpha properly with the minimum values for each equivalence class
- 0.30: Calling gradeAlpha properly with the maximum values of each equivalency class
- 0.20: Checking that the results obtained are correct
- Significant errors are subject to penalties

PROBLEM 3

Section 1 (1.25 points)

```
public void replaceZerosByPrevSum() {
   if(head != null) {
      if(head.getInfo() == 0) {
           head = head.getNext();
      }
      Node res = head;
      Node temp = head;
      int sum = 0;
      while (temp != null) {
         if (temp.getInfo() != 0) {
             sum += temp.getInfo();
         }
         else {
             res.setInfo(sum);
             res.setNext(temp.getNext());
             res = res.getNext();
             sum = 0;
         }
         temp = temp.getNext();
      }
   }
}
```

Section 2 (0.75 points)

```
public class Main {
    public static void main(String[] args) {
        LinkedList myList = new LinkedList(0);
        myList.insert(4);
        myList.insert(4);
        myList.insert(0);
        myList.insert(17);
        myList.insert(17);
        myList.insert(0);
        myList.insert(5);
        myList.insert(5);
        myList.insert(5);
        myList.print();
        System.out.println("\n");
        myList.replaceZerosByPrevSum();
        myList.print();
    }
}
```

Section 1 (1.25 points)

- 0.25: Loop to traverse the list
- 0.50: Sum of the nodes until some 0 is reached
- 0.50: Update of the auxiliary variables for the next cycle
- Significant errors are subject to penalties

Section 2 (0.75 point)

- 0.50: Building the list
- 0.25: Calling the method
- Significant errors are subject to penalties

PROBLEM 4 (2 points)

```
public boolean someSubtreeEqualsToSomeSubtreeOf(BTree<E> other) {
        return !other.isEmpty() &&
               (find(other)
                || someSubtreeEqualsToSomeSubtreeOf(other.getLeft())
                || someSubtreeEqualsToSomeSubtreeOf(other.getRight())
               );
}
```

- 0.50: Returning false when reaching the empty subtrees
- 0.50: Checking first if the whole other tree is the same as this one or any of the subtrees of this one (by calling this.find(other), or in some other way).
- 0.50: Correctly applying the same algorithm to the left and right subtrees of other trees.
- 0.50: Combining the subresults appropriately so that you start by looking for the largest possible subtree and continue to try smaller subtrees.
- Non-recourse solutions are allowed if they are correct.
- Penalty of -0.25 if, despite having already found a pair of identical subtrees, you continue to create tree partitions to find more pairs
- The solution will be valid both if the search is applied to this and partitions to other, or vice versa. The same with the empty tree check, which can be on this or on other.
- Significant errors are subject to penalties