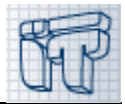


FIRST NAME:
LAST NAME:
NIA:
GROUP:

Second Part: Problems (7 points out of 10)

Duration: 180 minutes
Highest score possible: 7 points
Date: June 27, 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 (2.25 points)

A company that commercializes an e-mail service needs to implement an authentication system to access its service. The e-mail service offers two kinds of accounts: a general-purpose account that can be used for personal use, modeled in the class `Account`, and a professional account modeled in the class `ProfessionalAccount`.

Section 1.1 (0.75 points)

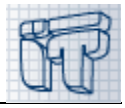
Implement the class that models the general-purpose account (`Account`), considering the following requirements:

- The class must store the username (`username`) and the password (`password`) for each account. Both attributes can contain alphanumeric characters. In addition, the class will contain an attribute to determine whether the account is blocked or not (`isBlocked`). These attributes cannot be directly accessed from any other class.
- The password must have a minimum length of 8 characters. This minimum length of 8 characters must be modeled with a constant (`PASSWORD_MIN_LENGTH`). If there is an attempt to set a password with a smaller length, a `PasswordException` will be thrown with the message *"The length of the password must be at least 8 characters"*.
- The class `PasswordException` is already implemented so you do not need to worry about implementing it.
- The constructor receives as parameters the username and the password. The password must fulfill the requirement of minimum length previously mentioned.
- The access methods (`get` and `set`) must be implemented considering that the values of all the attributes can be retrieved (`get`), but only the attributes that indicates if the account is blocked or not and the password (must always fulfill the minimum length restriction) can be modified (`set`).

Section 1.2 (0.5 points)

Implement the class that models the professional account (`ProfessionalAccount`), considering that this is a specific type of account (`Account`) with the following particularities:

- In addition to the attributes of `Account`, this class stores the name of the company (`company`) to which the user belongs. The company name will be initialized in the constructor and cannot be changed after the creation of the professional account, although its value can be retrieved from other classes.



- The password has the same minimum length restriction as in the case of the `Account` class, although it includes an additional security validation whereby the value of the password cannot be equal to the value of the username.

Section 1.3 (1 point)

Implement the class `AuthenticatorManager`, which keeps a list of e-mail accounts (regardless of whether the accounts are for personal or professional use), and handles the user authentication (checking that the username and password entered match those already stored in the list of e-mail accounts). The class `AuthenticatorManager` implements the following interface:

```
public interface Authenticator {

    static final int USER_AUTHENTICATED = 0;
    static final int PASSWORD_INCORRECT = 1;
    static final int ACCOUNT_BLOCKED = 2;
    static final int ACCOUNT_NOT_FOUND = 3;

    int authenticateUser(String username, String password);
}
```

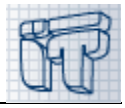
The class `AuthenticatorManager` must fulfill the following requirements:

- The constructor receives an array with all the accounts registered in the service (`Account[] accounts`).
- The method `authenticateUser(...)` is in charge of the authentication using the username and password received as parameters and the array of accounts received in the constructor. This method returns an integer indicating one of the 4 possibilities of the authentication process, as established in the interface `Authenticator`. The security is a very important aspect for the service provider, so when a user enters a wrong password once, the account must be blocked automatically.
- For the implementation of the method `authenticateUser(...)`, you must be considered that there will not be two accounts with the same username.

Problem 2 (0.75 points)

Given the class `Number`:

```
public class Number {
	private int number;
	public Number (int number){
		this.number = number;
	}
	public int result(){
		if (number > 0) {
			return 1;
		} else if (number < 0) {
			return -1;
		} else {
			return 0;
		}
	}
}
```



You are asked to implement the following tests. If a test cannot be implemented, then you must justify the why. **Note:** You can only create a maximum of one object per section and you cannot make several calls to the same method in the same section.

Section 2.1 (0.45 points)

Program a test which achieves a **branch coverage** between 1% (included) and 33% (not included).

Section 2.2 (0.15 points)

Program a test which achieves a **branch coverage** between 50% (not included) and 67% (included).

Section 2.3 (0.15 points)

Program a test which achieves a **method coverage** between 1% and 50% (both included).

Problem 3 (2 points)

The company *"Scheduled Packets"* is a delivery company dedicated to the distribution of packets between individuals. In order to organize and manage the reception and the shipment of the packets, the company has asked its team of programmers to create a software that facilitates and automates these tasks.

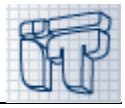
The task of the programming team is to implement a queue of packages to manage their reception and subsequent shipment. The team has some classes already programmed (see below) and assigns you the task of creating a new class, `PacketsQueue`, which extends `LinkedQueue<Packet>`. This new class will be the company's implementation of its own packet queue.

```java
public class Node<E>{
   private E info;
   private Node<E> next;
   public Node(){…}
   public Node(E info){…}
   public Node(E info, Node<E> next){…}
   public E getInfo(){…}
   public void setInfo(E info){…}
   public Node<E> getNext(){…}
   public void setNext(Node<E> next){…}
}
```

```java
public interface Queue<E>{
   boolean isEmpty();
   int size();
   E front();
   void enqueue (E info);
   E dequeue();
}
```

```java
public class Packet{
   private int numberId;
   private String dest;
   private boolean urgent;
   public Packet(int numberId, String dest, boolean urgent){
      this.numberId = numberId;
      this.dest = dest;
      this.urgent = urgent;
   }
   public int getNumberId(){ return numberId; }
   public void setNumberId(int numberId){
      this.numberId = numberId;
   }
   public String getDest(){ return dest; }
   }
   public void setDest(String dest){
      this.dest = dest;
   }
   public boolean isUrgent(){ return urgent; }
   public void setUrgent(boolean urgent){
      this.urgent = urgent;
   }
}
```

```java
public class LinkedQueue<E> implements
Queue<E>{
   protected Node<E> top;
   protected Node<E> tail;
   protected int size;

   public LinkedQueue(){…}

   public boolean isEmpty(){…}
   public int size(){…
   public E front(){…}
   public void enqueue (E info){…}
   public E dequeue(){…}
}
```



```
public class PacketsQueue extends LinkedQueue<Packet>{
  public PacketsQueue(){
        super();
  }

  public Packet sendPacket(){ //Section 3.1 }
  public Packet sendUrgent(){ //Section 3.2 }
}
```

Section 3.1 (0.4 points)

Program the method `public Packet sendPacket()`. This method returns the first packet that is waiting to be shipped and removes it from the queue. If there is no packet waiting to be shipped (the queue is empty), then the returned value must be `null`.

Section 3.2 (1.6 points)

Program the method `public Packet sendUrgent()`. This method returns the first **urgent** packet that is waiting to be shipped and removes it from the queue. If there is no urgent packet waiting to be shipped, the returned value must be `null`.

Problem 4 (2 points)

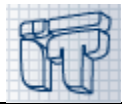
In binary trees, the **balance factor** of each node is defined as the difference between the height of its right subtree and the height of its left subtree, so that:

- A balance factor equals to 0 means that the two subtrees have the same height
- A positive balance factor indicates that the right subtree has a higher height than the left subtree.
- A negative balance factor indicates that the left subtree has a higher height than the right subtree.

In order to consider the balance factor, the attribute `private int balanceFactor` has been added to the `LBNode<E>` class, as well as the corresponding public getter and setter (`getBalanceFactor` and `setBalanceFactor` respectively), which are already implemented. Initially all the nodes will start from a balance factor with value 0, being necessary to call a method to update all the balance factors right after the creation of the tree has finished.

Add a method `public void updateBalanceFactor()` to the `LBTree<E>` class that updates in a recursive way the balance factor of every node of the tree.

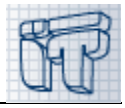
Next you can see the available methods of the interface `BTree<E>` and the implementation for the classes `LBNode<E>` and `LBTree<E>`:



```java
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);
}
```

```java
public class LBNode<E>{

  private E info;
  private BTree<E> left;
  private BTree<E> right;
  private int balanceFactor;

  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; }

  int getBalanceFactor() { return balanceFactor; }
  void setBalanceFactor(int balanceFactor){
    this.balanceFactor = balanceFactor;
  }
}
```

```java
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 void updateBalanceFactor() { //Problem 4 }
}
```



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

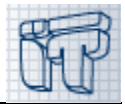
PROBLEM 1

Section 1.1 (0.75 points)

```
public class Account {
    private String username;
    private String password;
    private boolean isBlocked;

    protected static final short PASSWORD_MIN_LENGTH = 8;
    public Account(String username, String password) throws PasswordException {
        this.username = username;
        setPassword(password);
        isBlocked = false;
    }
    public String getUsername() {
        return username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) throws PasswordException {
        if (password.length() < PASSWORD_MIN_LENGTH) {
            throw new PasswordException("The length of the password must be at least
                8 characters");
        }
        this.password = password;
    }
    public boolean getIsBlocked() {
        return isBlocked;
    }
    public void setIsBlocked(boolean isBlocked) {
        this.isBlocked = isBlocked;
    }
}
```

Section 1.2 (0.5 points)

```
public class ProfessionalAccount extends Account {
    private String company;
    public ProfessionalAccount(String username, String password, String company)
            throws PasswordException {
        super(username, password);
        this.company = company;
    }

    public String getCompany() {
        return company;
    }
    @Override
    public void setPassword(String password) throws PasswordException {
        if (password.equals(getUsername())) {
            throw new PasswordException("The password cannot contain the username");
        }
        super.setPassword(password);
```



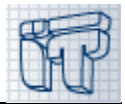
```
    }
}
```

Section 1.3 (1 point)

```
public class AuthenticationManager implements Authenticator {
    private Account[] accounts;

    public AuthenticationManager(Account[] accounts) {
        this.accounts = accounts;
    }

    public int authenticateUser(String username, String password) {
        Account account = null;
        boolean found = false;
        for (int i = 0; i < accounts.length && !found; i++) {
            if (accounts[i].getUsername().equals(username)) {
                account = accounts[i];
                found = true;
            }
        }
        int authenticationResult = PASSWORD_INCORRECT;
        if (account == null) {
            authenticationResult = ACCOUNT_NOT_FOUND;
        } else if (account.getIsBlocked()) {
            authenticationResult = ACCOUNT_BLOCKED;
        } else if (account.getPassword().equals(password)) {
            authenticationResult = USER_AUTHENTICATED;
        } else {
            account.setIsBlocked(true);
        }
        return authenticationResult;
    }
}
```

Section 1.1 (0.75 points)

- 0.05: Class declaration.
- 0.1: Correct attributes (type and visibility).
- 0.05: Correct constant (type and final). The assigned visibility will not be considered.
- 0.2: Constructor.
    - 0.05: Method definition including the exception.
    - 0.05: username initialization and isBlocked set to false (it is also correct if the line with "isBlock = false" is not included because isBlocked is false by default).
    - 0.1: Setting the password including the validation, invoking the set method or checking the length within the constructor.
- 0.1: Getter and setter (except setPassword).
- 0.25: setPassword.
    - 0.05: Method definition including the exception.
    - 0.1: Length verification.
    - 0.05: Throwing the exception.
    - 0.05: Setting the attribute.



Section 1.2 (0.5 points)

- 0.1: Class declaration.
  - -0.05 if it does not extend Account.
- 0.05: Attribute company and its corresponding getter.
- 0.15: Constructor.
  - 0.05: Constructor definition. If the exception is not defined here, but there was a penalty in Account, no additional penalty will be added.
  - 0.05: Calling superclass constructor (super).
  - 0.05: Initializing company.
- 0.2: setPassword.
  - 0.1: Checking username and exception.
  - 0.1: Calling superclass method.
  - If superclass set method is called before performing the validation of the username -0.1.

Section 1.3 (1 point)

- 0.1: Class declaration.
- 0.1: Attribute accounts.
- 0.15: Constructor.
- 0.65: Method authenticateUser.
  - 0.35: Searching for the account in the array.
    - -0.05 if == used instead of equals.
    - -0.15 if no loop is used to find the account.
  - 0.05: User not found condition.
  - 0.05: Blocked account condition.
  - 0.1: User correctly authenticated condition.
  - 0.1: Incorrect password condition and account blocking.
    - -0.05 if the account is not blocked but a value is returned (or the other way around).
  - If it does not return anything -0.1.

PROBLEM 2 (0.75 points)

Section 2.1 (0.45 points)

```
@Test
public void testSectionA(){
    Number number = new Number(5);
    assertEquals(number.result(),1);
}
```

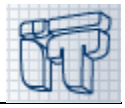
Section 2.2 (0.15 points)

```
// It cannot be done
```

It is not possible to achieve the requested coverage because:

1. Since there are 4 branches, coverage can only take values 0%, 25%, 50%, 75% or 100%, depending on the number of branches covered.
2. It is requested to achieve a coverage between 50% (not included) and 67% (included). Then, there is no possible way to achieve this range.

The 4 branches are: the true and false branches of the if (if number>0 or not) and the true and false branches of the else-if (if number<0 or not).



Section 2.3 (0.15 points)

```
@Test
public void testSectionC(){
    Number number = new Number(5);
}
```

Section 2.1 (0.45 points)

- 0.05: Correct header of the test method (with any method name):
- 0.1: Correct creation of the object of class Number: `Number number = new Number(5)`
- 0.3: Correct assertEquals.
- If code is correct but the test does not achieve the requested coverage, 0.2 for the whole section.

Section 2.2 (0.15 points)

- 0.15: If the reason why it cannot be done is properly justified. A possible justification is included in the solution above.
- If the answer is "It cannot be done" without justification 0.05 for the whole section.

Section 2.3 (0.15 points)

- 0.05: Correct header of the test method (with any method name):
- 0.1: Correct creation of the object of class Number: `Number number = new Number(5)`
- If code is correct but the test does not achieve the requested coverage, 0.05 for the whole section.

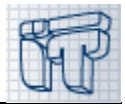
PROBLEM 3

Section 3.1 (0.4 points)

```
public Packet sendPacket(){
    return this.dequeue();
}
```

Section 3.2 (1.6 points)

```
public Packet sendUrgent(){
    if(top!=null){
        Node<Packet> tmp = top.getNext();
        Node<Packet> prev = top;
        if(prev.getInfo().isUrgent()){
            return this.dequeue();
        }else{
            while(tmp!=null){
                if(tmp.getInfo().isUrgent()){
                    prev.setNext(tmp.getNext());
                    size--;
                    if(tmp.getNext()==null){
                        tail = prev;
                    }
                    return tmp.getInfo();
                }else{
                    prev = tmp;
                    tmp = tmp.getNext();
```



```
                }
            }
        }
    }
    return null;
}
```

Section 3.1 (0.4 points)

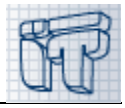
- 0.4: Correct handling of the dequeue operation, manually of through the dequeue() method:
  Manually:
  - 0.1: Updating top.
  - 0.1: Updating tail.
  - 0.1: Updating size.
  - 0.1: Returning the packet.

  Through the dequeue() method:
  - 0.3: Calling the dequeue() method.
  - 0.1: Returning the packet.

Section 3.2 (1.6 points)

- 0.4: Returning null when there are no urgent packets.
- 0.4: Correct handling of the case in which the first urgent packet is referenced by top, manually or though the dequeue() method:
  Manually:
  - 0.1: Updating top.
  - 0.1: Updating tail.
  - 0.1: Updating size.
  - 0.1: Returning the packet.

  Through the dequeue() method:
  - 0.3: Calling the dequeue() method.
  - 0.1: Returning the packet.
- 0.4: Correct handling of the case in which the first urgent packet is in the middle of the queue:
  - 0.2: Updating next references.
  - 0.1: Updating size.
  - 0.1: Returning the packet.
- 0.4: Correct handling of the case in which the first urgent packet is referenced by tail:
  - 0.1: Updating next references.
  - 0.1: Updating tail.
  - 0.1: Updating size.
  - 0.1: Returning the packet.



PROBLEM 4 (2 points)

```
public void updateBalanceFactor() {
	if (!isEmpty()) {
		this.root.setBalanceFactor(this.getRight().height() -
		this.getLeft().height());
		((LBTree<E>) this.getLeft()).updateBalanceFactor();
		((LBTree<E>) this.getRight()).updateBalanceFactor();
	}
}
```

- 0.25: Determining the base case condition.
- 0.25: Solving the base case (no modification should be done when the tree is empty).
- 0.50: Correct calculation of the balance factor for the own node (computing the subtraction between the height of the right subtree and the height of the left one). If calculated in the opposite way (left height – right height), maximum 0.4.
- 0.50: Storing the factor using the method setBalanceFactor of the object this.root. If the private attribute is used and not the setter, maximum 0.40.
- 0.25: Recursive call of the method for the left subtree.
- 0.25: Recursive call of the method for the right subtree.
- Penalize -0.05 if no casting is performed to call the recursive method (if only one is missing or both of them).