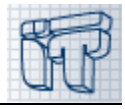


FIRST NAME:
LAST NAME:
NIA:
GROUP:

First midterm exam

Second Part: Problems (7 points out of 10)

Duration: 70 minutes
Highest score possible: 7 points
Date: March 18, 2022

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 (4 / 7 points)

A tyre manufacturer is developing an advanced type of tyres to obtain the best performance. The performance depends on the height (`height`) and the working temperature (`temp`) of the tyre. The manufacturer has designed three different types of tyres so far, which are summer, winter and all-season tyres. For summer tyres it is also relevant to know if they can run flat (`runFlat`) or not, and the wear coefficient (`wear`). For winter tyres it is important to know the snow grip coefficient (`snowGrip`). These coefficients can be expressed as `double` numbers.

The manufacturer considers 25 degrees as the normal temperature for testing the tyres. This temperature is used to calculate the optimum working temperature for each type of tyre, which differs from one type of tyre to another, according to the following:

- For summer tyres, if they can run flat the optimum temperature will be 2.5 times the normal temperature. Otherwise it will be the normal temperature multiplied by the wear coefficient.

- For winter tyres the optimum temperature will be the normal temperature multiplied by the snow grip coefficient.

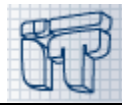
- For all-season tyres, if the height is greater than 50 the optimum temperature will be 80.3 degrees. Otherwise it will be 75.7.

Section 1.1 (0.4 points)

Declare the interface `AdvancedTyreInterface` containing a constant for the normal temperature and the method `calculateOptimumTemp()`.

Section 1.2 (0.8 points)

Write the code for the class `AdvancedTyre`. Declare all the attributes and the getter and setter for the attribute `height`. Also write the code for a constructor with all the attributes. Before assigning the value of the attribute `height` you must check the value to be assigned is positive, and launch the exception `NegativeNumberException` otherwise. Assume this exception is already created for you. The class `AdvancedTyre` must implement the interface declared in the first section.



Section 1.3 (1.2 points)

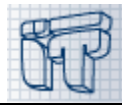
Declare the classes `SummerTyre`, `WinterTyre` and `AllSeasonTyre`. Declare the attributes and write the code for the method `public double calculateOptimumTemp()` for each class. You do not need to declare the getters and setters for these classes. Assume getters and setters are created for you if you need to use them.

Section 1.4 (0.8 points)

Write the code for the testing method `public void testCalculateOptimumTemp()` to test the method `public double calculateOptimumTemp()` in class `AllSeasonTyre`. The test must meet these two conditions: On the one hand the test must pass (positive test), and on the other hand it must have 100% line coverage on the method `public double calculateOptimumTemp()`.

Section 1.5 (0.8 points)

Write the code for the testing method `public void testSetHeight()` to test that the `NegativeNumberException` is thrown in the setter for the attribute `height`. Use the class `SummerTyre` for this purpose.



Problem 2 (3 / 7 points)

A leading logistics company has asked you to develop a program for managing a warehouse. To define Provider management, define two classes:

- Person `(Person)`. It refers to any relevant person for the management of a warehouse. It can be a customer of the warehouse, the employee that manages the order, or the contact person for a provider. To identify a person, we need his/her identification number (`id`), first name (`firstName`), last name (`lastName`) and contact email (`email`).

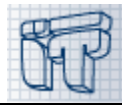
```
public class Person {
    private int id;
    private String firstName;
    private String lastName;
    private String email;

public Person(int id, String firstName, String lastName, String email) {
        try {
            setId(id);
            setFirstName(firstName);
            setLastName(lastName);
            setEmail(email);
        } catch (PersonException pe) {
            pe.printStackTrace();
            System.exit(1);
        }

public String toString() {
        return id + "|" + firstName + "|" + lastName + "|" + email +
"\n";
    }

}
```

Note: All getters and setters of this class are implemented.

You are asked to:

Section 2.1 (1 point)

Class `Provider` is the company in charge of supplying a product to the warehouse. It contains the company's tax identification number `(vat)`, its name `(name)`, the billing address `(taxAddress)` and a contact person `(contactPerson)`. This last attribute is of type `Person`. Implement the class `Provider` with its constructor, attributes only visible in this class, and a method `toString()` to print on screen all its attributes.



Section 2.2 (1 point)

Class `Worker` that inherits from the class `Person` which represents the workers in the warehouse. Each worker will have a salary and a position (only visible in this class):
All workers receive the same salary, this quarter it is 2.000 euros, although it is not fixed because it can increase each quarter if there are large profits in the business.
The worker's position will be modelled by means of constants that can take the following values: `WAREHOUSE = "stockman"; SELLER = "sales".`
Implement a constructor and assign the corresponding value to the attributes. You will need to check that the position is correct and if it is not, assign the default value that is `WAREHOUSE`. Implement also a method `toString()` to print on screen all its attributes.

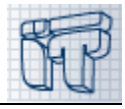
Section 2.3 (1 point)

Program a class `DatabaseWarehouse`, which will be the main class of your program, in which you will test your code. To do this, include a method `main`.
In that method you should initialize an `ArrayList <Person>` in which you will include a `Worker` and a `contactPerson` of Provider.
Also, you must call the method `toString()` that you have implemented in the previous sections, on the entries of your `ArrayList` so that the result that is printed on screen.

NOTE 1: You don't have to worry about importing the class `ArrayList`.

NOTE 2: The class `ArrayList` has the following methods, some of which may be useful:

• `boolean add(E e)` • `void add(int index, E element)` • `void clear()` • `E get(int index)`	• `int indexOf(Object o)` • `boolean isEmpty()` • `boolean remove(Object o)` • `int size()`



REFERENCE SOLUTIONS (several solutions are possible)

PROBLEM 1

Section 1.1 (0.4 points)

```
public interface AdvancedTyreInterface {

	public static final double NORMAL_TEMPERATURE = 25;
	public abstract double calculateOptimumTemp();
}
```

Evaluation criteria:

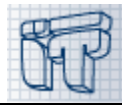
- 0 if the code makes no sense or the interface has any code different from declarations.
- 0.1 for declaring the interface.
- 0.15 for declaring the constant.
- 0.15 for declaring the abstract method.
- Significant errors are subject to additional penalties.

Section 1.2 (0.8 points)

```
public abstract class AdvancedTyre implements AdvancedTyreInterface {

	private double height;
	private double temp;

	public AdvancedTyre(double height, double temp) {
		try {
			setHeight(height);
		} catch (NegativeNumberException e) {
			e.printStackTrace();
		}
		this.temp = temp;
	}

	public double getHeight() {
		return height;
	}

	public void setHeight(double height) throws NegativeNumberException {
		if (height < 0)
			throw new NegativeNumberException("Invalid positive number");

		this.height = height;
	}
}
```

Evaluation criteria

- 0 if the code makes no sense.
- 0.1 for declaring the class as abstract and implementing the interface.
- 0.1 for declaring the attributes (0 if not declared as private or protected).
- 0.3 if the constructor is OK. If the exception is not caught nor thrown, maximum 0.1
- 0.3 if the setter and the getter are OK (if the exception is not thrown maximum 0.1).
- Significant errors are subject to additional penalties.

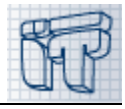


Section 1.3 (1.2 points)

```
public class SummerTyre extends AdvancedTyre {

	private boolean runFlat;
	private double wear;

	public double calculateOptimumTemp() {

		double result;

		if (runFlat) {
			result = NORMAL_TEMPERATURE * 2.5;
		} else
			result = NORMAL_TEMPERATURE * wear;

		return result;
	}
}

public class WinterTyre extends AdvancedTyre {

	private double snowGrip;

	public double calculateOptimumTemp() {

		return NORMAL_TEMPERATURE * snowGrip;
	}
}


public class AllSeasonTyre extends AdvancedTyre {

	public double calculateOptimumTemp() {

		double result;

		if (getHeight() > 50) {
			result = 80.3;
		} else
			result = 75.7;

		return result;
	}
}
```

Evaluation criteria:

- 0 if the code makes no sense.
- 0.4 for each class (0.1 for declaring the class and the attributes, and 0.3 if the method calculateOptimumTemp is OK).
- Constructors are optional
- Significant errors are subject to additional penalties.



Section 1.4 (0.8 points)

```
@Test
public void testCalculateOptimumTemp() {

	AllSeasonTyre tyre = new AllSeasonTyre(55,
			AdvancedTyreInterface.NORMAL_TEMPERATURE);

	assertEquals(tyre.calculateOptimumTemp(), 80.3, 0.1);

	AllSeasonTyre tyre2 = new AllSeasonTyre(45,
			AdvancedTyreInterface.NORMAL_TEMPERATURE);

	assertEquals(tyre2.calculateOptimumTemp(), 75.7, 0.1);
}
```

Evaluation criteria

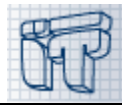
- 0 if the code makes no sense.
- 0.1 for the annotation.
- 0.1 for properly creating the object from class AllSeasonTyre with the right values (or setting them after the creation).
- 0.25 for each call to the method assertEquals (each call will cover one branch of the if-else structure). If the precision argument is missing in calling the method assertEquals, maximum 0.2 for each call.
- 0.1 for creating a second object, or altering the height of the first object in order to test the second branch.
- Significant errors are subject to additional penalties.

Section 1.5 (0.8 points)

```
@Test
public void testSetHeight() {

// Two possibilities:

   // Junit4
	try {
		SummerTyre tyre = new SummerTyre();
		tyre.setHeight(-1);
		fail("Failed test");
	} catch (NegativeNumberException e) {
		e.printStackTrace();
	}

	// JUnit5:
	assertThrows(NegativeNumberException.class,   ()->{SummerTyre   tyre   =   new
SummerTyre(); tyre.setHeight(-1);});
}
```

Evaluation criteria

- 0 if the code makes no sense.
- 0.1 for the annotation.
- 0.1 for properly creating the object from class SummerTyre.
- 0.2 for setting the height to a value which will throw the exception.
- 0.4 for properly calling assertThrows, or using the try-catch block and calling method fail inside it.
- Significant errors are subject to additional penalties.



PROBLEM 2

Section 2.1. (1 point)

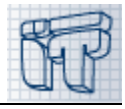
```
public class Provider {
	private int vat;
	private String name;
	private String taxAddress;
	private Person contactPerson;


	public Provider (int vat, String name, String taxAddress, Person c) {
		this.vat = vat;
		this.name = name;
		this.taxAddress = taxAddress;
		contactPerson = c;
	}
	public String toString() {
			return vat + "|" + name + "|" + taxAddress + "|" +
			contactPerson.toString();
	}
}
```

Evaluation criteria

- 0. If the code makes no sense
- 0.1 Defines vat, name, taxAddress OK
- 0.15 Defines the attribute contactPerson OK
- 0.35 Defines constructor OK
- 0.1 Defines toString with String as a return type
- 0.2 Implements method toString() well with attributes vat, name and taxAddress
- 0.1 Calls attribute contactPerson in method toString() OK
  Significant errors are subject to additional penalties.

Section 2.2 (1 point)

```
public class Worker extends Person {

	private static int salary = 2000;
	private String job;
	private final static String WAREHOUSE = "stockman";
	private final static String SELLER = "sales";

    public Worker (int id, String firstName, String lastName, String email,
    String job) {
        super(id,firstName, lastName, email);
        if (!job.equals(WAREHOUSE) && !job.equals(SELLER)) {
		job = WAREHOUSE;
        }
		this.job = job;
        }

    public String toString() {
        return salary + "|" + job + "|" + super.toString();

        }
}
```



Evaluation criteria:

- 0 If the code makes no sense
- 0.2 Defines attribute job, salary OK
- 0.2 Defines WAREHOUSE and SELLER OK
- 0.4 Implements constructor OK
    - 0.2 Assign id, firstName, email OK
    - 0.2 Assign attribute job OK
- 0.2 Defines method toString() well
    - 0.1 Attributes salary, job
    - 0.1 Call to method toString()
- Significant errors are subject to additional penalties.

Section 2.3 (1 point)

```
public class DatabaseWarehouse {

public static void main(String[] args) {
            Person a = new Person(2,"John", "Miller", "john@gmail.com");
            Worker t1 = new Worker(1, "Eva","Garcia",
            "eva@gmail.com","stockman");
            Provider c1 = new Provider(3, "Ivan", "Cliente 1",a);

            ArrayList<Person> list = new ArrayList<Person>();
             list.add(t1);
             list.add(c1.getContactPerson());

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

Evaluation criteria:

- 0.5 Defines all the objects OK
    - 0.15 Object Person,
    - 0.15 Object Worker,
    - 0.2 Object Provider
- 0.1 Creates ArrayList <Person> OK
- 0.1 Adds object Worker to ArrayList OK
- 0.1 Adds attribute contactPerson of Provider to ArrayList() OK
- 0.2 Implements the method toString(). OK
    - The toString() in the get(i).toString() is optional (no necessary)
- Significant errors are subject to additional penalties.