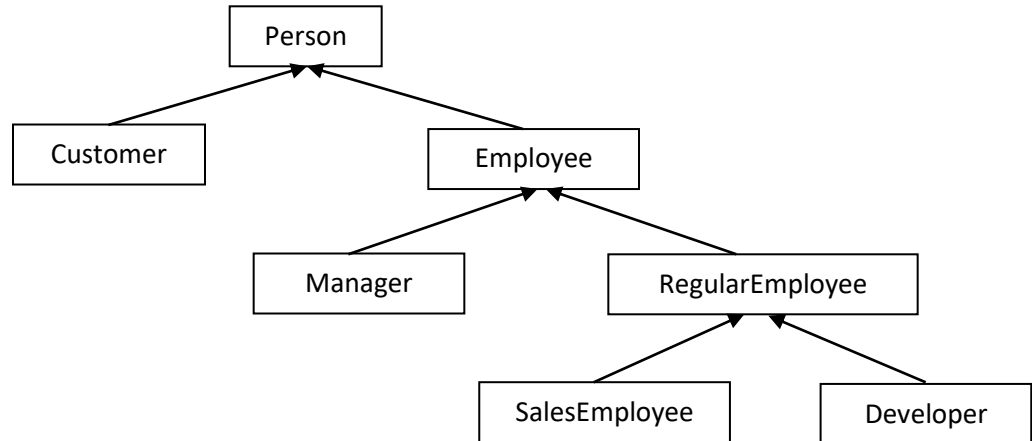


CSE 1142 - COMPUTER PROGRAMMING II
Programming Assignment # 1
DUE DATE: 11/03/2019 - 23:00 (No extension)

In this homework, you are expected to implement a simple company system with the following OOP class hierarchy:



Please find the class details in below.

1. Implement a **Person** class with the following UML diagram.

Person	
-	id: int
-	firstName: String
-	lastName: String
-	gender: byte
-	birthDate: java.util.Calendar
-	maritalStatus: byte
-	hasDriverLicence: boolean
+	Person(int id, String firstName, String lastName, String gender, Calendar birthDate, String maritalStatus, String hasDriverLicence)
	setGender(gender: String):void
+	getGender():String
+	setMaritalStatus(status: String)
+	getMaritalStatus(): String
+	setHasDriverLicence (info: String): void
+	getHasDriverLicence():String
+	toString(): String
+	getter/setter methods for other data fields
+	

- **Person** is the superclass of **Customer** and **Employee** classes.
- **Person** class has several data fields, getter/setter and **toString** methods.

- Each person should have an **id**, a **name**, a **surname**, a **gender** (1: woman, 2: man), **birthDate** (05/05/2000), **maritalStatus** (1: single, 2: married) and **hasDriverLicense** attributes.
- Since the parameter/return value types are different for getter/setter methods of **gender**, **maritalStatus**, and **hasDriverLicence** attributes, we show them in the UML diagram. You are responsible for implementing getter/setter methods of all data fields.
- In **setGender** method, a string value ("Man" or "Woman") is given, and the method should set the **gender** as 1 or 2.
- In **getGender** method, a string value ("Man" or "Woman") should be returned based on the **gender** value.
- In **setMaritalStatus** method, a string value ("Single" or "Married") is given, and the method should set the **maritalStatus** as 1 or 2.
- In **getMaritalStatus** method, a string value ("Single" or "Married") should be returned based on the **maritalStatus** value.
- In **setHasDriverLicence** method, a string value ("Yes" or "No") is given, and the method should set the **hasDriverLicence** as true or false.
- In **getHasDriverLicence** method, a string value ("Yes" or "No") should be returned based on the **hasDriverLicence** value.
- There are setter/getter and toString() methods.

2. Implement a **Customer** class with the following UML diagram.

Customer	
-	products: ArrayList<Product>
+	Customer(int id, String firstName, String lastName, String gender, Calendar birthDate, String maritalStatus, String hasDriverLicence, ArrayList<Product> products)
+	Customer(Person person, ArrayList<Product> products)
+	getter/setter methods
+	toString(): String

- Each **Customer** can be created with one of the given two constructors.
 - In Customer's constructor, you are supposed to call the super class's constructor.
- Each **Customer** has a list of **products** that he/she bought.
- There are setter/getter and toString() methods.

3. Implement an **Employee** class with the following UML diagram.

Employee	
-	salary: double
-	hireDate: java.util.Calendar
-	department: Department
+	<u>numberOfEmployees</u> : int
+	Employee(int id, String firstName, String lastName, String gender, Calendar birthDate, String maritalStatus, String hasDriverLicence, double salary, Calendar hireDate, Department department)
+	Employee(Person person, double salary, Calendar hireDate, Department department)
+	raiseSalary(percent: double):double
+	raiseSalary (amount: int): double
+	getter/setter / toString methods

- **Employee** is the superclass of **Manager** and **RegularEmployee** classes.
- Each **Employee** has a **salary**, a **hireDate** (the date when the employee starts to the job), a **department** and **numberOfEmployees** data fields.
- Each **Employee** can be created with one of the given two constructors.
 - In **Employee**'s constructor, you are supposed to call the super class's constructor.
 - When a new employee is created, you should increment the value of **numberOfEmployees** by 1.
- There are two overloaded implementations of **raiseSalary** method.
 - In the first one, take the increment value as a double ($0 \leq \text{percent} \leq 1$) and raise the salary value based on the percentage value. For example, if the percent value is 0.5, increment the salary of the employee by 50%.
 - In the second one, raise the salary of the employee by the given fixed amount.
- There are setter/getter and toString() methods.

4. Implement a **RegularEmployee** class with the following UML diagram.

RegularEmployee	
-	performanceScore: double
-	bonus: double
+	RegularEmployee(int id, String firstName, String lastName, String gender, Calendar birthDate, String maritalStatus, String hasDriverLicence, double salary, Calendar hireDate, Department department, double performanceScore)
+	RegularEmployee(Employee employee, double perfScore)
+	getter/setter and toString methods

- **RegularEmployee** is the superclass of **SalesEmployee** and **Developer** classes.
- Each **RegularEmployee** has a **performanceScore** and an amount of **bonus**, which will be given by his/her manager based on the performance score.
- Each **RegularEmployee** can be created with one of the given two constructors.
 - In **RegularEmployee** 's constructor, you are supposed to call the super class's constructor.
- There are setter/getter and toString() methods.

5. Implement a **Manager** class with the following UML diagram.

Manager	
-	regularEmployees: ArrayList<RegularEmployee>
-	bonusBudget: double
+	Manager(int id, String firstName, String lastName, String gender, Calendar birthDate, String maritalStatus, String hasDriverLicence, double salary, Calendar hireDate, Department department, double bonusBudget)
+	Manager(Employee employee, double bonusBudget)
+	addEmployee (e: RegularEmployee): void
+	removeEmployee(e: RegularEmployee): void
+	distributeBonusBudget():void
+	getter/setter / toString methods

- Each **Manager** has a set of **regularEmployees** working in his/her department and a **bonusBudget** to distribute to the regular employees in the department.
- Each **Manager** can be created with one of the given two constructors.
 - In **Manager**'s constructor, you are supposed to call the super class's constructor.
- In **addEmployee** method, you should add the given **RegularEmployee e** to the list of **regularEmployees**.
- In **removeEmployee** method, you should remove the given **RegularEmployee e** from the list of **regularEmployees**.
- Each **Manager** has **bonusBudget** to distribute it to the regular employees working in his/her department. The distribution will be based on the given formula:
 - Suppose that the bonus budget of the manager is 10000 and there are 4 regular employees in the department with the following salaries and performance scores:
 - E1 → salary: 1000, performanceScore:50
 - E2 → salary: 2000, performanceScore:50
 - E3 → salary: 6000, performanceScore:75
 - E4 → salary: 4000, performanceScore:100
 - Then, the bonus value of each regular employee is:
 - **bonus = unit * salary * performanceScore**
 - **unit = bonusBudget / $\sum(\text{salary} * \text{performanceScore})$**

- Based on the example above, the bonus value for each regular employee is:
 - E1 → bonus: 500
 - E2 → bonus: 1000
 - E3 → bonus: 4500
 - E4 → bonus: 4000
- There are setter/getter and toString() methods.

6. Implement a **SalesEmployee** class with the following UML diagram.

SalesEmployee	
-	sales: ArrayList< Product>
+	<u>numberOfSalesEmployees</u> : int
+	SalesEmployee(int id, String firstName, String lastName, String gender, Calendar birthDate, String maritalStatus, String hasDriverLicence, double salary, Calendar hireDate, Department department, double pScore, ArrayList<Product> s)
+	SalesEmployee(RegularEmployee re, ArrayList<Product> s)
+	addSale (s: Product): boolean
+	removeSale(s: Product): boolean
+	getter/setter / toString methods

- Each **SalesEmployee** has a set of **sales** that contains a product list that the **SalesEmployee** sells and a **numberOfSalesEmployees** data fields.
- Each **SalesEmployee** can be created with one of the given two constructors.
 - In **SalesEmployee**'s constructor, you are supposed to call the super class's constructor.
 - When you create a new **SalesEmployee**, you should increment **numberOfSalesEmployees** value by 1.
- In **addSale** method, you should add the given **Product s** to the list of **sales**.
- In **removeSale** method, you should remove the given **Product s** from the list of **sales**
- There are setter/getter and toString() methods.

7. Implement a **Developer** class with the following UML diagram.

Developer	
-	projects : ArrayList< Project >
+	<u>numberOfDevelopers</u> : int

+	Developer(int id, String firstName, String lastName, String gender, Calendar birthDate, String maritalStatus, String hasDriverLicence, double salary, Calendar hireDate, Department department, double pScore, ArrayList<Project> p)
+	Developer(RegularEmployee re, ArrayList<Project> p)
+	addProject (s: Project): boolean
+	removeProject(s: Product): boolean
+	getter/setter / toString methods

- Each **Developer** has a set of **projects** that the developer works on and a **numberOfDevelopers** data fields.
- Each **Developer** can be created with one of the given two constructors.
 - In **Developer**'s constructor, you are supposed to call the super class's constructor.
 - When you create a new **Developer**, you should increment **numberOfDevelopers** value by 1.
- In **addProject** method, you should add the given **Projects s** to the list of **projects**.
- In **removeProject** method, you should remove the given **Product s** from the list of **projects**.
- There are setter/getter and toString() methods.

8. Implement a **Product** class with the following UML diagram.

Product	
-	productName: String
-	saleDate: java.util.Calendar
-	price: double
+	Product(String sName, java.util.Calendar sDate, double price)
+	getter/setter / toString methods

- Each **Product** has a **name**, **saleDate** and **price** data fields.
- There are setter/getter and toString() methods.

9. Implement a **Project** class with the following UML diagram.

Project	
-	projectName: String
-	startDate: java.util.Calendar
-	state: boolean
+	public Project (String pName, Calendar startDate, String state)
+	setState(state: String): void
+	getState(): String
+	close(): void
+	getter/setter / toString methods

- Each **Project** has a **name**, **startDate** and **state** data fields. If the **Project** is open, **state** should be true; otherwise, false.
- In **setState** method, a string value ("Open" or "Close") is given, and the method should set the **state** as true or false.
- In **getState** method, a string value ("Open" or "Close") should be returned based on the **state** value.
- In **close** method, you should close the project if it is open.
- There are setter/getter and toString() methods.

10. Implement a **Department** class with the following UML diagram.

Department	
-	departmentId: int
-	departmentName: String
+	Department(int departmentId, String departmentName)
+	getter/setter / toString methods

- Each **Department** has an **id** and a **name** data fields.
- There are setter/getter and toString() methods.

11. Implement a test class for your program.

- You should read input from a file and create new objects based on the line read. A set of sample lines in your input file is given below:
 - **Department 1 Accounting**
 - You should create a new **Department** with an id of **1** and name of **Accounting**.
 - **Project AutoCredit 01/05/2018 Open**
 - You should create a new **Project** with the name of **AutoCredit**, startDate **01/05/2018** and state **Open**.
 - **Product Product1 01/01/2019 10000**
 - You should create a new **Product** with the name of **Product1**, saleDate **01/01/2019** and price of **10000**.
 - **Person Ayse Caliskan 111 Woman 05/05/1986 Married Yes**
 - You should create a new **Person** with the name of **Ayse**, surname **Caliskan**, id of **111**, gender **1**, birth date **05/05/1986**, maritalStatus **2** and hasDriverLicence **true**.
 - **Employee 111 5000 10/10/2017 Accounting**
 - You should create a new **Employee** by finding the **Person** with the given id (111) and invoke the overloaded constructor of **Employee** with the **Person** found, salary: **5000**, hireDate: **10/10/2017** and department: **Accounting**.

- **RegularEmployee 111 25**
 - You should create a new **RegularEmployee** by finding the **Employee** with the given id (111) and invoke the overloaded constructor of **RegularEmployee** with the **Employee** found and performanceScore **25**.
 - **Developer 111 CreditCard Robotic**
 - You should create a new **Developer** by finding the **RegularEmployee** with the given id (111) and invoke the overloaded constructor of **Developer** with the **RegularEmployee** found, project list: **CreditCard** and **Robotics**. It should be noted that the number of projects may change.
 - **Customer 224 Product1 Product2 Product5**
 - You should create a new **Customer** by finding the **Person** with the given id (224) and invoke the overloaded constructor of **Customer** with the **Person** found, product list: **Product1** **Product2** and **Product5**. It should be noted that the number of products may change.
- b. After reading the input file and constructing the objects (keep your objects in **ArrayLists** polymorphically in the test file), the following sample scenario can be given in your test class:
- i. invoke **distributeBonusBudget** method for each **Manager** polymorphically.
 - ii. invoke **raiseSalary** method for each **Manager** polymorphically with the percent value of **0.2**.
 - iii. invoke **raiseSalary** method for each **RegularEmployee** polymorphically with the percent value of **0.3**.
 - iv. invoke **raiseSalary** method for each **Developer** polymorphically with the percent value of **0.23**.
 - v. invoke **raiseSalary** method for each **SalesEmployee** polymorphically with the percent value of **0.18**.
 - vi. invoke **raiseSalary** method for a **SalesEmployee** who has maximum value of sales (in terms of total price) polymorphically with the amount of **1000**.
- c. After performing these operations, print each department, it's manager, it's employee list with details. A sample output file is already generated based on the given sample input file and the execution scenario mentioned in Step b.

This is a simple scenario to test your class implementations. There might be other test cases too. Therefore, please pay attention to use the same class, method and variable names in your implementations. You are allowed to increase the number of methods in the classes; however, you cannot decrease the number of them.

Several Important Notes:

- All properties' values should not be blank.
- All integer type values in the setters, should be validated to be positive only.
- All string type values in the setters, should be validated to be no less than 3 symbols.
- If you enter invalid input for one of the properties' values, throw **Exception** (not **RuntimeException**!) with an informative message.
- It should be noted that selected parts will be graded in your solution.

Submission Instructions:

Please zip and submit your files using filename YourNumberHW1.zip (ex: 150713852HW1.zip) to Canvas system (under Assignments tab). Your zip file should contain the following files:

1. 11 Java source files: Person.java, Customer.java, Employee.java, Manager.java, RegularEmployee.java, SalesEmployee.java, Developer.java, Product.java, Project.java, Department.java, Test.java.
2. 11 Java .class files
3. An input file: input.txt
4. An output file: output.txt

Notes:

1. Write a comment at the beginning of your program to explain the purpose of the program.
2. Write your name and student ID as a comment.
3. Include necessary comments to explain your actions.
4. Select meaningful names for your variables and class name.
5. You are allowed to use the materials that you have learned in lectures & labs.
6. Do not use things that you did not learn in the course.
7. **Program submissions** should be done through the Canvas class page, under the assignments tab. Do not send program submissions through e-mail. E-mail attachments will not be accepted as valid submissions.
8. You are responsible for making sure you are turning in the right file, and that it is not corrupted in anyway. We will not allow resubmissions if you turn in the wrong file, even if you can prove that you have not modified the file after the deadline.
9. In case of any form of **copying and cheating** on solutions, you will get **FF** grade from the course! You should submit your own work. In case of any forms of cheating or copying, both giver and receiver are equally culpable and suffer equal penalties. **All types of plagiarism will result in FF grade from the course.**
10. No late submission will be accepted.