Assignment: Object-Oriented Programming

Exercise 1 – Basic Object-Oriented Principles

Objective

You will develop a Java program that focuses on the following key principles of object-oriented programming:

- **Polymorphism**: Implement different behaviors of a single operation (makeSound) across a hierarchy of classes.
- **Constructors**: Implement default and parameterized constructors for each class and demonstrate constructor chaining.
- **Class Variables**: Use class-level (static) variables to track the number of instances created for specific classes.
- **Interfaces**: Implement an interface that enforces a common behavior across multiple classes.
- **Abstract Classes**: Use an abstract class to enforce method overriding in subclasses.

Instructions

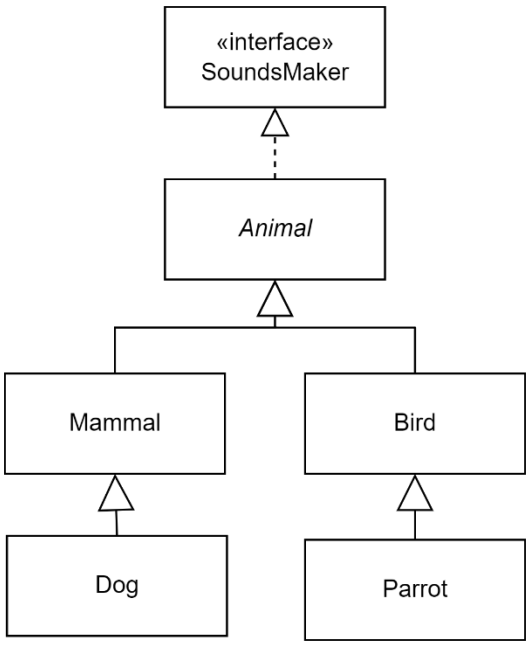

Figure 1 – Class Hierarchy for Animal

For this exercise, you will create the class hierarchy shown in Figure 1, which consists of the `SoundsMaker` interface, the abstract class `Animal`, and the classes `Mammal`, `Bird`, `Dog`, and `Parrot`. Follow the step-by-step instructions provided to implement the hierarchy.

Step 1: Define the Interface `SoundMaker`

1. Create an interface called `SoundMaker`.
2. In the `SoundMaker` interface, define the following method signature:

```
void makeSound();
```

   This interface will be used to enforce that all animals must have a `makeSound()` method.

Step 2: Define the Abstract Base Class `Animal`

1. **Create an abstract class** called `Animal` that implements the `SoundMaker` interface.
2. **Fields:**
   a. Create a **static** (class-level) variable `numberOfAnimals` to track the total number of animals created.
   ```
   static int numberOfAnimals = 0;
   ```
   b. Create a non-static field `name` of type `String` to store the name of the animal.
3. **Constructors:**
   a. Implement a **default constructor** that:
      - Increments the `numberOfAnimals` by 1.
      - Prints `"Animal Constructor called"`.
      - Initializes the `name` field to `"Unknown Animal"`.
   b. Implement a **parameterized constructor** that:
      - Accepts a `String` parameter for the animal's name.
      - Increments the `numberOfAnimals` by 1.
      - Prints `"Animal Constructor with name called"`.
      - Initializes the `name` field with the provided value.
4. **Methods:**
   a. Define an **abstract method** `getAnimalType()` which will return a `String` representing the type of animal.
   b. Implement a method `getNumberOfAnimals()` that returns the total number of animals created.

Step 3: Define the Subclass `Mammal`

1. **Create a class** called `Mammal` that extends `Animal`.
2. Create a **static** variable `numberOfMammals` to track the total number of mammals created.
3. **Constructors:**
   a. Implement a **default constructor** that:
      - Calls the **default constructor** of `Animal` (Hint: lookup the keyword `super`).
      - Increments `numberOfMammals` by 1.

        - Prints `"Mammal Constructor called"`.
    b. Implement a **parameterized constructor** that:
        - Accepts a `String` parameter for the mammal's name.
        - Calls the **parameterized constructor** of `Animal`.
        - Increments `numberOfMammals` by 1.
        - Prints `"Mammal Constructor with name called"`.
4. **Methods**:
    a. Implement the `makeSound()` method from the `SoundMaker` interface. This method should print:
    `System.out.println("Mammal sound");`
    b. Implement the `getAnimalType()` method to return:
    `return "Mammal";`
    c. Implement a method `getNumberOfMammals()` that returns the total number of mammals created.

Step 4: Define the Subclass `Bird`

1. **Create a class** called `Bird` that extends `Animal`.
2. **Fields**:
    a. Create a **static** variable `numberOfBirds` to track the total number of birds created.
3. **Constructors**:
    a. Implement a **default constructor** that:
        - Calls the **default constructor** of `Animal`.
        - Increments `numberOfBirds` by 1.
        - Prints `"Bird Constructor called"`.
    b. Implement a **parameterized constructor** that:
        - Accepts a `String` parameter for the bird's name.
        - Calls the **parameterized constructor** of `Animal` using `super(name)`.
        - Increments `numberOfBirds` by 1.
        - Prints `"Bird Constructor with name called"`.
4. **Methods**:
    a. Implement the `makeSound()` method from the `SoundMaker` interface.
    b. Implement the `getAnimalType()` method.
    c. Implement a method `getNumberOfBirds()` that returns the total number of birds created.

Step 5: Define the Subclass `Dog` (Subclass of `Mammal`)

1. **Create a class** called `Dog` that extends `Mammal`.
2. **Constructors**:
    a. Implement a **default constructor** that:
        - Calls the **default constructor** of `Mammal`.
        - Prints `"Dog Constructor called"`.
    b. Implement a **parameterized constructor** that:
        - Accepts a `String` parameter for the dog's name.
        - Calls the **parameterized constructor** of `Mammal`.
        - Prints `"Dog Constructor with name called"`.

3. **Override** the `makeSound()` and `getAnimalType()` methods.

Step 6: Define the Subclass `Parrot` (Subclass of `Bird`)

1. **Create a class** called `Parrot` that extends `Bird`.
2. **Constructors**:
    a. Implement a **default constructor** that:
        - Calls the **default constructor** of `Bird`.
        - Prints `"Parrot Constructor called"`.
    b. Implement a **parameterized constructor** that:
        - Accepts a `String` parameter for the parrot's name.
        - Calls the **parameterized constructor** of `Bird`.
        - Prints `"Parrot Constructor with name called"`.
3. **Override** the `makeSound()` and `getAnimalType()` methods.

Step 7: Demonstrate Polymorphism, Constructor Chaining, and Class Variables

In the `main` method of a `AnimalTest` class:

a. **Create lists** (e.g. array lists) for each type of animal (`Animal`, `Mammal`, `Bird`, `Dog`, `Parrot`)
b. **For each list**, create and add **two instances** using default constructors and **two instances** using parameterized constructors (total of **four instances** per class).

   HINT: `Animal` is an abstract class. Therefore, you cannot create instances of `Animal`. Hence, you should add an instance of each of the other classes to the `Animal` list.

c. **Iterate** over each list and call the `makeSound()` method on every object.
d. **After all instances are created**, print the total number of created instances for each class by calling their respective static methods (i.e., `getNumberOfAnimals()`, `getNumberOfMammals()`, `getNumberOfBirds()`).

Expectations

Make sure to use the appropriate access modifiers (private, protected, public) for variables and methods to promote encapsulation.

Add comments to your code so that JavaDocs can be generated. Include descriptions for classes, constructors, methods, and fields.

Appendix 1 provides the program's sample output.

Exercise 2 – University Management System

Objective

A key skill in software engineering is having the ability to add features to an existing system. Hence, the objective of this exercise is to push you out of your comfort zone by presenting you with a moderately complex system (compared to the systems from your first-year programming courses). In this exercise, you will read and understand aspects of someone else's code. Additionally, you will add simple features to the existing system.

Instructions

Exercise 2.1

For this exercise, you are provided with a basic University Management System. This system allows the user to create students, employees (e.g., administrators and professors), and courses. Additionally, it allows the user to register a student to a course and assign a professor to a course. The system has a Command Line Interface whose commands are provided in the Appendix 2.

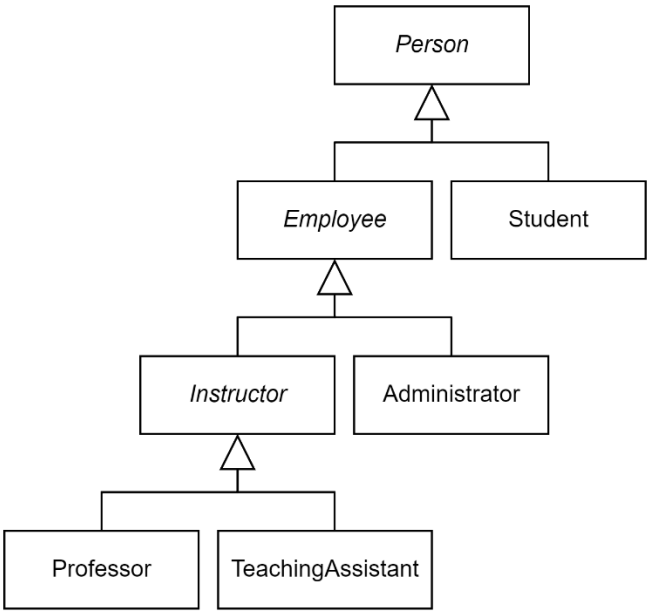

Figure 2 – UML Hierarchy for Person Entities

Step 1: Download and Run the System

Download the Eclipse Project that accompanies this assignment and run the program. Familiarize yourself with the commands described in the Appendix 2. For instance, you may create a course, create a student, create a professor, assign a professor to a course, and register a student to a course.

Step 2: Add the TA Functionalities

The entities that represent people (i.e. students, administrators, and professors) are represented in the hierarchy described in the UML diagram of Figure 2. **The** `TeachingAssistant` **class is not implemented yet. Implement the** `TeachingAssistant` **class by extending the** `Instructor` **class.** While a professor can teach 5 courses simultaneously, make sure that a TA can only have a workload of 3 courses at a time.

Step 3: Update the `Course` class

Update the `Course` class so that a TA can be assigned to a course. For simplicity, there can only be one TA for each course. Update the `toString()` method in the course to generate a string that includes the name of the teaching assistant (similar to what is already done for the professor).

Step 4: Update the `UniversityManagementSystem` Class

1. Study the following methods carefully in the `UniversityManagementSystem` Class:
   - `public void processCreateEmployee(String entity, List<String> input)`
   - `public void processAssignInstructor(List<String> input)`
   - `public void processListEmployees(String entity)`
2. Update the methods listed above to support the following commands:
   - `create TA "<First name>" "<Last name>" "<salary>"`
   - `assign TA "<employee id>" "<course code>"`
   - `list TAs`
   - `get TA "<employee id>"`

   You do not need to add code to the `UniversityManagementSystem` class to support the **get command** as all `Employee` instances are fetched in the same manner regardless of the type of employee.

   Note: This update does not require any changes to the `CommandLineInterface` class code as these commands are already supported by the user interface.

Exercise 2.2

Although it is currently possible to create and get an administrator entity and list all stored administrators, it is not possible to assign an administrator a task.

Step 1: Complete the implementation of the `Administrator` class

Complete the functionality of the `Administrator` class so that an administrator can be assigned tasks. Leverage the existing `tasks` list. You will need to add at least three methods to the `Administrator` class:

Method	Description
`public void addTask(String task):`	Adds a task to the administrator.
`private String getTasksList()`	A method that should be invoked by the `toString()` method to return a list of the administrator's tasks in the for of a `String` where each responsibility is presented on a separate line.
`public String toString()`	Generates a String representation of the administrator object (see other entity classes for examples). It invokes `getTasksList()` to get a list of tasks to be included in the returned `String`.

Step 2: Update the `UniversityManagementSystem` Class

Complete the code for the `processAssignAdministrator` method in the *`UniversityManagementSystem`* class so that an administrator can be assigned tasks. Once the updates are completed, the system should be able to process the following command:

```
assign administrator "<employee id>" "<work responsibility>"
```

Note: This update does not require any changes to the `CommandLineInterface` class code as the command is already supported by the user interface.

Appendix 1

Sample output for the Exercise 1 Program:

```
Animal Constructor called
Mammal Constructor called
Animal Constructor called
Bird Constructor called
Animal Constructor with name called
Mammal Constructor with name called
Dog Constructor with name called
Animal Constructor with name called
Bird Constructor with name called
Parrot Constructor with name called
Animal Constructor called
Mammal Constructor called
Animal Constructor called
Mammal Constructor called
Animal Constructor with name called
Mammal Constructor with name called
Animal Constructor with name called
Mammal Constructor with name called
Animal Constructor called
Bird Constructor called
Animal Constructor called
Bird Constructor called
Animal Constructor with name called
Bird Constructor with name called
Animal Constructor with name called
Bird Constructor with name called
Animal Constructor called
Mammal Constructor called
Dog Constructor called
Animal Constructor called
Mammal Constructor called
Dog Constructor called
Animal Constructor with name called
Mammal Constructor with name called
Dog Constructor with name called
Animal Constructor with name called
Mammal Constructor with name called
Dog Constructor with name called
Animal Constructor called
Bird Constructor called
Parrot Constructor called
Animal Constructor called
Bird Constructor called
```

```
Parrot Constructor called
Animal Constructor with name called
Bird Constructor with name called
Parrot Constructor with name called
Animal Constructor with name called
Bird Constructor with name called
Parrot Constructor with name called

Animals making sounds:
Mammal sound
Bird sound
Woof!
Squawk!

Mammals making sounds:
Mammal sound
Mammal sound
Mammal sound
Mammal sound

Birds making sounds:
Bird sound
Bird sound
Bird sound
Bird sound

Dogs making sounds:
Woof!
Woof!
Woof!
Woof!

Parrots making sounds:
Squawk!
Squawk!
Squawk!
Squawk!

Total number of animals: 20
Total number of mammals: 10
Total number of birds: 10
```

Appendix 2

Full list of commands supported by the university management system, along with their required parameters and usage:

1. **Create Commands**
   These commands allow the creation of various entities within the university management system, including courses, professors, administrators, and students.

   - `create course "<course code>" "<course title>" "<course description>"`
     Creates a new course in the system with a unique course code, description, and specified capacity.

     Example usage:
     `create course SEG2105 "Introduction to Software Engineering" "Principles of software engineering: Requirements, design and testing. Review of principles of object orientation. Object oriented analysis using UML. Frameworks and APIs. Introduction to the client-server architecture. Analysis, design and programming of simple servers and clients. Introduction to user interface technology."`

   - `create professor "<First name>" "<Last name>" "<salary>"`
     Creates a new professor entity with the provided first name, last name, and salary.

     Example usage:
     `create professor "John" "Doe" "85000"`

   - `create administrator "<First name>" "<Last name>" "<salary>"`
     Creates a new administrator entity with the given first name, last name, and salary.

   - `create student "<First name>" "<Last name>" "<program of study>"`
     Creates a new student entity with the given first name, last name, a program of study.

2. **Assign Commands**

   These commands are used to assign professors and students to courses, as well as administrators to their responsibilities.

   - `assign professor "<employee id>" "<course code>"`
     Assigns a professor with the specified employee ID to a course using the course code.

Example usage:
assign professor "0001234" "SEG2105"

- `assign student "<student id>" "<course code>"`
  Registers a student with the provided student ID to a course using the course code.

3. **List Commands**

The list commands display all instances of the specified entity type in the system.

- `list employees`
  Displays all employees (including professors and administrators) currently stored in the system.

- `list administrators`
  Displays all administrators in the system.

- `list professors`
  Displays all professors in the system.

- `list students`
  Displays all students in the system.

- `list courses`
  Displays all courses available in the system.

4. **Get Commands**

The get commands allow users to retrieve detailed information for a specific entity based on their unique ID or course code.

- `get employee "<employee id>"`
- `get administrator "<employee id>"`
- `get professor "<employee id>"`
- `get student "<student id>"`
- `get course "<course code>"`