Lab 3 - Extension of Name Management Considering Realistic Student Numbers

This lab introduces inheritance and advanced object-oriented programming concepts through three progressive tasks. Students will learn class inheritance by extending the Person class to create a Student class, practice polymorphism and method overriding, and work with scalable data structures for realistic applications. The exercises progress from student management systems to vehicle hierarchies and animal classification, emphasizing inheritance relationships, code reusability, and polymorphic behavior.

Contents

1. Task 1: Student Management with Inheritance

Extend your Java program from Lab 2 to demonstrate inheritance by creating a Student class that inherits from the Person class. This task introduces class inheritance, scalability, and advanced object management.

Extend your name management program as follows:

- Create a new `Student` class that inherits all aspects of the `Person` class
- Add a matriculation number attribute that starts at 1001 and increments automatically
- Use a constant `MAX_ANZAHL` to define maximum students (default: 500, scalable to millions)
- Implement proper encapsulation for the matriculation number (private with getter/setter)
- Extend the menu system:
  - Input 0: Exit program
  - Input `MAX_ANZAHL+1`: Display all student attributes
  - Input `MAX_ANZAHL+2`: Display maximum manageable students

Detailed requirements:

- Create a new class Students that inherits all aspects of the Person class in addition to the matriculation number. The matriculation number starts with 1001 and increases by one with each additional person. This matriculation number can also only be accessed via appropriate methods. Otherwise, everything remains as before, i.e., the program is terminated with input 0, with input `MAX_ANZAHL+1` all attributes of all students are displayed, and with `MAX_ANZAHL+2` the maximum number of students manageable by the program is output.

1.1. Preparation

First clarify the task by drawing the Student and Person classes according to UML notation, considering their inheritance relationship. Then define all necessary classes, methods, and variables in Java.

1.2. Assistance

The following code snippets demonstrate key inheritance concepts:

Basic inheritance structure:

```java
public class Person {
    protected String firstName;
    protected String lastName;
    protected int day, month, year;
    private static int personCount = 0;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        personCount++;
    }

    // Getter and setter methods...
}

public class Student extends Person {
    private int matriculationNumber;
    private static int nextMatriculationNumber = 1001;

    public Student(String firstName, String lastName) {
        super(firstName, lastName); // Call parent constructor
        this.matriculationNumber = nextMatriculationNumber++;
    }

    public int getMatriculationNumber() {
        return matriculationNumber;
    }
}
```

Using constants for scalability:

```java
public class StudentManager {
    private static final int MAX_ANZAHL = 500;
    private Student[] students = new Student[MAX_ANZAHL];

    // Menu options based on MAX_ANZAHL
    if (choice == MAX_ANZAHL + 1) {
        displayAllStudents();
    } else if (choice == MAX_ANZAHL + 2) {
        System.out.println("Maximum students: " + MAX_ANZAHL);
```

```
10      }
11  }
```

2. Task 2: Vehicle Hierarchy with Polymorphism

Create a vehicle management system that demonstrates inheritance and polymorphism. This task will help you understand method overriding and polymorphic behavior.

Create a class hierarchy for different types of vehicles:

- Base class `Vehicle` with common properties (brand, model, year)
- Subclasses `Car`, `Motorcycle`, and `Truck` that extend Vehicle
- Each subclass should have specific attributes and override methods appropriately

Your program should:

1. Create a base `Vehicle` class with:
   - Protected attributes: brand, model, year
   - Constructor and getter methods
   - A `displayInfo()` method that shows basic vehicle information
   - An abstract or overridable `calculateMaintenanceCost()` method
2. Create subclasses that:
   - Add specific attributes (e.g., `numberOfDoors` for Car, `cargoCapacity` for Truck)
   - Override `displayInfo()` to include subclass-specific information
   - Implement `calculateMaintenanceCost()` with different formulas
3. Create an array of Vehicle objects containing different types
4. Use polymorphism to call methods on all vehicles in the array

2.1. Requirements

- Use inheritance with `extends` keyword
- Demonstrate method overriding with `@Override` annotation
- Create polymorphic behavior using Vehicle array containing different subclasses
- Use `protected` access modifiers appropriately

Note about `@Override`: The `@Override` annotation is a helpful tool that tells Java you intend to override a method from the parent class. While not strictly required, it's considered good practice because it helps catch errors at compile time. If you misspell a method name or get the parameters wrong, Java will give you an error instead of accidentally creating a new method.

2.2. Assistance

Basic vehicle hierarchy:

```java
public abstract class Vehicle {
    protected String brand;
    protected String model;
    protected int year;

    public Vehicle(String brand, String model, int year) {
        this.brand = brand;
        this.model = model;
        this.year = year;
    }

```

```
12      public void displayInfo() {
13          System.out.println(year + " " + brand + " " + model);
14      }
15
16      public abstract double calculateMaintenanceCost();
17  }
18
19  public class Car extends Vehicle {
20      private int numberOfDoors;
21
22      public Car(String brand, String model, int year, int doors) {
23          super(brand, model, year);
24          this.numberOfDoors = doors;
25      }
26
27      @Override
28      public void displayInfo() {
29          super.displayInfo();
30          System.out.println("Doors: " + numberOfDoors);
31      }
32
33      @Override
34      public double calculateMaintenanceCost() {
35          return (2025 - year) * 150.0; // Older cars cost more
36      }
37  }
```

Polymorphic array usage:

```
1   Vehicle[] vehicles = {                                          Java
2       new Car("Toyota", "Camry", 2020, 4),
3       new Motorcycle("Honda", "CBR", 2021, 600),
4       new Truck("Ford", "F-150", 2019, 1500)
5   };
6
7   for (Vehicle v : vehicles) {
8       v.displayInfo(); // Calls overridden method
9       System.out.println("Maintenance: $" + v.calculateMaintenanceCost());
10  }
```

3. Task 3: Animal Classification System

Create an animal classification system that demonstrates advanced inheritance concepts including abstract classes and interfaces. This task focuses on designing hierarchical relationships.

Design a system that models different types of animals with their behaviors:

- Abstract base class `Animal` with common properties
- Interface `Flyable` for animals that can fly
- Interface `Swimmable` for animals that can swim
- Concrete animal classes that implement appropriate interfaces

Your program should:

1. Create an abstract `Animal` class with:
   - Protected attributes: name, species, age
   - Abstract method `makeSound()`
   - Concrete method `displayInfo()`
2. Create interfaces:
   - `Flyable` with method `fly()`
   - `Swimmable` with method `swim()`
3. Create concrete animal classes:
   - `Bird` (extends Animal, implements Flyable)
   - `Fish` (extends Animal, implements Swimmable)
   - `Duck` (extends Animal, implements both Flyable and Swimmable)
   - `Dog` (extends Animal, no additional interfaces)
4. Demonstrate multiple inheritance through interfaces

3.1. Requirements

- Use abstract classes and methods
- Implement multiple interfaces in same class
- Override abstract methods in concrete classes
- Demonstrate interface polymorphism

3.2. Assistance

Abstract class and interface structure:

```java
public abstract class Animal {
    protected String name;
    protected String species;
    protected int age;

    public Animal(String name, String species, int age) {
        this.name = name;
        this.species = species;
        this.age = age;
    }

    public void displayInfo() {
```

```
13          System.out.println(name + " is a " + age + " year old " + species);
14      }
15
16      public abstract void makeSound();
17  }
18
19  public interface Flyable {
20      void fly();
21      default int getMaxAltitude() { return 1000; } // Default method
22  }
23
24  public interface Swimmable {
25      void swim();
26  }
27
28  public class Duck extends Animal implements Flyable, Swimmable {
29      public Duck(String name, int age) {
30          super(name, "Duck", age);
31      }
32
33      @Override
34      public void makeSound() {
35          System.out.println("Quack!");
36      }
37
38      @Override
39      public void fly() {
40          System.out.println(name + " is flying over the pond!");
41      }
42
43      @Override
44      public void swim() {
45          System.out.println(name + " is swimming in the water!");
46      }
47  }
```

4. Lab Execution:

If your program is not yet working without issue, we will try to correct this during the course of the lab. With good preparation, this should not be a problem. Every student is required to be able to explain their thought process at the beginning of the lab. By the end of the lab, the task needs to be completed. Of course, we will support you, but your personal commitment must also be clearly recognizable! Julian Moldenhauer, Furkan Yildirim, and Emily Antosch wish you lots of fun and success!