Lab 2 for Object-Oriented Programming and Design: Inheritance, Polymorphism and Abstract Classes

Compiled by Pakizar Shamoi, PhD, Professor
Kazakh-British Technical University

October 6, 2024

Contents

Introduction: Laboratory Rules

This lab is designed to improve your OOP coding skills and deepen your understanding of key OOP concepts of inheritance, polymorphism, and abstract classes. Please carefully follow the guidelines below.

Key Rules and Guidelines:

- **Deadline.** The deadline for this lab is the **8th week**, practice time.
- **Prohibition on AI-generated Code. The use of AI-generated code (e.g., ChatGPT or other similar tools) is strictly prohibited.** If AI-generated code is found in your lab, even in a single problem, you will receive **0 points** for the entire lab. You are expected to write and develop the code on your own.
- **Points.** Each problem costs 2 points, so, 10 points total.
- **Class Requirements.** Make sure you have read and understood them.
- **Naming Conventions.** It is critical to follow them for all your Java classes, methods, and variables (camel case). Consistent naming conventions are essential for clean and readable code.
- **Separation of Classes.** Each class should be kept in its own file. This ensures better organization and clarity. Each problem should have a test class in which you test your classes.
- **Recreate the Code.** The purpose of this lab is for you to practice and learn, so you must recreate the code by yourself. Btw, showing your real, authentic code is much better than attempting to cheat or copy from external sources.
- **Hints.** Code hints are given to you so you can have an easy start and initial structure. Due to this, the size of this lab seems big :)

Final Note

This lab is for your growth as a developer. Take it seriously, put in your best effort, and most importantly, **learn from the process**. Success doesn't come from shortcuts but from persistence and practice. Be persistent, please, even when the challenges seem difficult. Every problem you solve is a step towards becoming a better software engineer.

Good luck, dear students!

Problem 1: Warm-up tasks

Part a: 3D Shapes

Create an abstract class for 3D shapes with methods `volume()` and `surfaceArea()`. Then, create data types `Cylinder`, `Sphere`, and `Cube` that extend this class. **Hints:**

```
public abstract class Shape3D {
    public abstract double volume();
    ...
    // continue
}

public class Cylinder extends Shape3D {
    private double radius, height;
    public Cylinder(double r, double h) {
        this.radius = r;
        this.height = h;
    }
    @Override
    public double volume() {
        return Math.PI * radius * radius * height;
    }
     ...
    // continue
}
    // continue with other classes
```

Part b: Library System

Design a system for managing a library's collection. You have different types of items, such as books, DVDs, and magazines. Each item has some common properties (e.g., title, author, and publication year), but also specific properties. Create an abstract class `LibraryItem` and **one of the** subclasses (choose one!) such as `Book`, `DVD`, and `Magazine`, inheriting from `LibraryItem`. implement the specific properties and methods for the subclass you chose. **Hints:**

```
public abstract class LibraryItem {
    private String title;
    private String author;
    private int publicationYear;
```

```
    // Constructor, Getters, and Setters
    // toString() and other methods
}

public class Book extends LibraryItem {
    private int numberOfPages;
    public Book(String title, String author, int year, int pages) {
        super(title, author, year);
        this.numberOfPages = pages;
    }
    // continue with anything necessary
}
```

Part c: Superclass and Subclass

Create a superclass and subclass of your choice. Override `equals()` and `hashcode()` methods in both classes. Test the classes by adding objects to a `HashSet` and verifying no duplicate entries. ***This is an example, you have to think of another example.***

```
// Superclass definition
public class Person {
    private String name;
    private int age;
    // Getters, Setters, toString, equals, hashCode methods
}
// Subclass definition
public class Employee extends Person {
    private String employeeId;
    // Override equals and hashcode methods
    @Override
    public boolean equals(Object o) {
        // Custom equality check
    }
    @Override
    public int hashCode() {
        // Custom hash code logic
    }
} // do not forget test class
```

Problem 2: Chess Pieces

Create a data type for chess pieces. Inherit from the abstract class `Piece`, then create subclasses `Rook`, `King`, `Queen`, `Bishop`, `Knight`, and `Pawn`. Implement the `isLegalMove()` method for checking valid moves - that determines whether the given piece can move from `Position` a to b. Yes, `Position` is a class. Use it if you can't play chess - Chess Wikipedia Page. **Hints:**

```
public abstract class Piece {
    Position a;
    public abstract boolean isLegalMove(Position b);
    // anything else you need, continue
}

public class Rook extends Piece {
    @Override
    public boolean isLegalMove(Position b) {
        // Check horizontal or vertical moves
    }
        // anything else you need, continue
        //continue with other classes
}
```

Bonus: Chess Game Implementation

Create a class `Board` and test class to manage the current state of a chess game and fully imitate the game. Implement methods to handle user input, draw the board on the console, drawing the board on a console, checking for illegal moves, etc.

Bonus costs 1-4 points, depending on your result. Bonus needs to be defended separately during office hours, only to Pakita or your lecturer.

Problem 3: Bank Account System

Write the `Account` class and use it as a base class, then write two derived classes called `SavingsAccount` and `CheckingAccount`.

SavingsAccount (aka "deposit"):

- In addition to the attributes of an `Account` object, it should have an interest rate variable (%) and a method to add interest to the account.

CheckingAccount (there is a charge for each transaction):

- In addition to the attributes of an `Account` object, it should have a counter variable to store the number of transactions done by the user.
- Define `FREE_TRANSACTIONS` as the number of free transactions. Also, include a method `deductFee()` that withdraws $0.02 for each transaction (withdrawal or deposit) exceeding the free transactions.

Ensure that you override the necessary methods of the `Account` class in both derived classes. **Hints:**

```
public class Account{
    private double balance; // The current balance
    private int accNumber; // The account number
    public Account(int a){
        balance = 0.0;
        accNumber = a;
    }
    public void deposit(double sum) { ,,, }

    public void withdraw(double sum) { ,,, }

    public double getBalance() { ,,, }

    public double getAccountNumber() { ,,, }

    public void transfer(double amount, Account other) { ,,, }

    public String toString() { ,,, }

    public final void print() {
        // Don't override this, override the toString method
        System.out.println(toString());
    }
}
```

Bank Class

- Create a `Bank` class, which contains a `Vector` of `Account` objects.
- The `Accounts` in the `Vector` can be instances of `Account`, `SavingsAccount`, or `CheckingAccount`.

- Write an `update` method that iterates through each account and processes transactions:
  - `SavingsAccount` objects get interest added.
  - `CheckingAccount` objects get fees deducted.
- Create test accounts (some of each type).
- Implement methods for opening and closing accounts.

Problem 4: Electrical Circuit System

Or a nightmare for IS students :) When electricity moves through a wire, it is subject to electrical friction or *resistance*. When a resistor with resistance `R` is connected across a potential difference `V`, Ohm's law asserts that it draws current `I = V / R` and dissipates power `V^2 / R`. A network of resistors connected across a potential difference behaves as a single resistor, which we call the *equivalent resistance*.

You should have an abstract superclass `Circuit` that encapsulates the basic properties of a resistor network. For example, each network has a method `getResistance` that returns the equivalent resistance of the circuit.

```
public abstract class Circuit {
    public abstract double getResistance();
    public abstract double getPotentialDiff();
    public abstract void applyPotentialDiff(double V);
    public double getPower() {
        // your code
    }
    public double getCurrent() {
        // your code
    }
}
```

A series-parallel resistor network is either (i) a single resistor or (ii) built up by connecting two resistor networks in series or parallel. So create three `Circuit` class subclasses: `Resistor`, `Series`, and `Parallel`.

Class `Resistor`:

- The class `Resistor` contains a constructor that sets the resistance in Ohms and an accessor method to return it.
- It also has a private field `potentialDifference` with get/set methods.

Class `Series`:

- The class `Series` contains a constructor that takes two circuit objects as inputs and represents a circuit with the two components in series.

Class `Parallel`:

- The class `Parallel` is similar to `Series`, but it uses the reciprocal rule instead of the additive rule to compute the equivalent resistance.

Your goal is to be able to compose circuits as in the following code fragment, which represents the circuit depicted below.

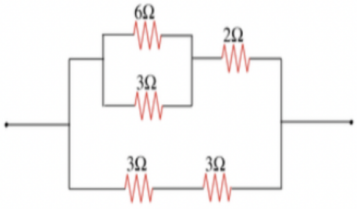

Figure 1: Illustration of a circuit given in an example

```
Circuit a = new Resistor(3.0);
Circuit b = new Resistor(3.0);
Circuit c = new Resistor(6.0);
Circuit d = new Resistor(3.0);
Circuit e = new Resistor(2.0);
Circuit f = new Series(a, b);
Circuit g = new Parallel(c, d);
Circuit h = new Series(g, e);
Circuit circuit = new Parallel(h, f);
double R = circuit.getResistance();
```

Additional Notes:

- The potential difference across each section depends on whether the circuit is series or parallel.
- For a parallel circuit, the potential difference across each branch is equal to the potential difference across the whole parallel circuit.
- For a series circuit, first find the current `I` from Ohm's law `I = V / R`, where `V` is the potential difference across the series circuit and `R` is its total resistance. The potential difference across each resistor is equal to the current times the resistance of that resistor.

Problem 5: Pet Management System

You are tasked with creating a **Pet Management System** that models different types of people (`Employee`, `Student`, `PhDStudent`) and pets. In this system, people may or may not own pets, which can be various animals like cats, dogs, or birds. In addition, sometimes people need to leave their pets with others in the system, when they go on vacation.

- **Person Class (abstract)**:
  - Each person has `name`, `age`, and possibly a pet (`Animal`).
  - Methods:
    - `assignPet(Animal pet)`: Assigns a pet to a person.
    - `removePet()`: Removes the pet from a person.
    - `hasPet()`: Checks if the person has a pet.
    - `getOccupation()`: Describes the person's occupation (to be implemented by subclasses).
    - Additional methods like `toString()`, `equals()`, `hashCode()`.
- **Employee, Student, PhDStudent Classes**:
  - These extend `Person`, each with its specific fields and implementing `getOccupation()`. Each subclass should have at least 1 new field different from its parent. PhD students are so consumed with their research that they can't have high-maintenance pets like dogs, they are too busy to provide constant attention. `PhDStudent` can have more independent pets like `Cat`, `Fish`, and `Bird`. The system should not allow `PhDStudent` to have a `Dog`; other people can't leave their dogs to `PhDStudent` as well.
- **Animal Class (abstract)**:
  - Every animal has a `name` and `age`, and each specific type of animal (e.g., `Cat`, `Dog`, `Bird`, `Fish`) has its way of making a sound via the `getSound()` method (to be implemented by subclasses).
- **Animal Subclasses**:
  - Implement concrete animals like `Cat`, `Dog`, `Fish` and `Bird`, each providing its specific sound when `getSound()` is called.
- **PersonRegistry Class**:

- A registry to hold a collection of people and allow operations like adding/removing people, finding people with/without pets, and printing details about each person and their pets.

- **Vacation Logic**:
  - People can leave their pets with others while they are on vacation. Implement a way for a person to temporarily transfer their pet to another person.
  - The system should ensure that, when a person leaves their pet, the pet is removed from their care and assigned to the temporary caretaker. When the person returns, they should be able to retrieve their pet.
  - You'll need to consider how to handle cases where a person doesn't have a pet, but tries to leave one with a friend or a service.
  - Consider how to model the temporary transfer of a pet and its return after a vacation.

Sample Interaction:

```
Person john = new Employee("John", 30, "Engineer");
Person alice = new PhDStudent("Alice", 26, "Comp. Science", "AI");
Animal murka = new Cat("Murka", 5);
john.assignPet(murka); // John owns Rex
PersonRegistry registry = new PersonRegistry();
registry.addPerson(john);
registry.addPerson(alice);
// John goes on vacation and leaves Rex with Alice
john.leavePetWith(alice);
// Registry reflects that Alice is taking care of Rex
System.out.println(registry);
// John returns from vacation and retrieves Rex
john.retrievePetFrom(alice);
// Registry reflects that John has his dog back
System.out.println(registry);
```