Exercise 5

Objectives

The aim of this exercise is to learn the concept of inheritance. In this exercise you will learn:

- The concept of **inheritance**
- The concept of **composition** (transfer/delegation).
- How to construct a class hierarchy

Problem A

Inheritance: MovingPoint2D [2 pt]

You have a Point2D class that you developed for problem D in ex02. First, extend this class by adding the following methods.

Method Name	Details
`distance(Point2D other)`	Find the distance between itself and a given Point2D object and return it as a real number.
`toString()`	Override the toString method of Class Object to return its coordinate in the form `(x, y)`

Now we need a new MovingPoint2D Class, which should have the following states and behaviors in addition to those possessed by the Point2D Class.

Field Name	Details
`vx`	Real number representing the velocity in the x direction.
`vy`	Real number representing the velocity in the y direction.

Method Name	Details
`setVelocity(double a, double b)`	Update its vx and vy with the given real numbers a and b, respectively.
`move()`	Move itself by adding vx and vy to the current location x and y, respectively.

You should also define a constructor to accept and initialize the current coordinate and the velocity.

Your task in this problem is to develop the MovingPoint2D Class by **Inheritance** of the Point2D Class.

Instances of MovingPoint2D should be manipulated by executing MovingPoint2DApplication Class shown below.

```
class MovingPoint2DApplication{
    public static void main(String[] args){
        MovingPoint2D p1 = new MovingPoint2D(0, 0, 3, 4);  // x, y, vx, vy
        MovingPoint2D p2 = new MovingPoint2D(0, 0, 0, -1);

        for ( int i = 0; i < 10; i++ ) {
            p1.move();
            p2.move();
        }

        System.out.println(p1 + "-" + p2 + ": " + p1.distance(p2));

        p2.setVelocity(30, 0);
        p2.move();

        System.out.println(p1 + "-" + p2 + ": " + p1.distance(p2));
    }
}
```

If MovingPoint2DApplication produces the following sample outputs, it can be judged that the MovingPoint2D class has been implemented correctly.

```
(30.0, 40.0)-(0.0, -10.0): 58.309518948453004
(30.0, 40.0)-(30.0, -10.0): 50.0
```

Submission Files	Types
A/Point2D.java	Java Class
A/MovingPoint2D.java	Java Class
A/MovingPoint2DApplication.java	Java Class

Problem B

Composition: MovingPoint2D [2 pt]

Your situation is similar to Problem A, with the need for MovingPoint2D. However, consider not employing inheritance this time.

Your task in this problem is to develop the MovingPoint2D Class by **Composition** (also called transfer/delegation in some contexts) of objects created by the Point2D Class.

In Composition, a new class is composed of objects of existing classes and you can reuse the functionality of the code.

Test your MovingPoint2D class with the MovingPoint2DApplication created in Problem A to see if it produces the same output. Note that the Point2D and MovingPoint2DApplication created in Problem A must not be modified.

Submission Files	Types
`B/Point2D.java`	`Java Class`
`B/MovingPoint2D.java`	`Java Class`
`B/MovingPoint2DApplication.java`	`Java Class`

Problem C

Constructor Chain: Trees [2 pt]

You understand how important tree structures are for retrieving information and implementing data structures. You are trying to develop a data structure related to binary trees. Examine the class hierarchy of the following types of trees and implement a Class that represents each tree.

`OrderedTree` `Tree` `BinaryTree` `RootedTree`

Each Class should only have a default constructor that outputs its Class name. Test your program with the following ConstructorChainApplication. **This test application should output the name of each class in the order in which its constructor is called** (i.e., the output should consist of four lines, each containing the name of a Class).

```
class ConstructorChainApplication{
   public static void main(String[] args){
      new BinaryTree();
   }
}
```

* This problem is an exercise in the importance of satisfying the "is-a" relationship when designing with inheritance. Note that in practice, designing a set of classes for such a data structure will require more sophisticated relationships and implementations.

Submission Files	Types
C/ConstructorChainApplication.java	Java Class
C/BinaryTree.java	Java Class
C/OrderedTree.java	Java Class
C/RootedTree.java	Java Class
C/Tree.java	Java Class

Problem D

Overriding: ModInt [2 pt]

Develop Int Class and MInt Class to handle integers. The Int Class should have the following fields and methods.

Field Name	**Details**
`x`	An integer that this object holds.
`INT_MAX`	The maximum integer value this class can represent, INT_MAX = 1024.

Method Name	**Details**
`add(Int a)`	Returns a new Int object obtained by adding the given Int object `a` to itself.
`mul(Int a)`	Returns a new Int object obtained by multiplying the given Int object `a` to itself.
`setValue(int a)`	Update `x` with the given integer (primitive type) `a`.
`getValue()`	Return the value of `x`.
`toString()`	Override the toString method of Class Object to return the value of `x`.

Develop the MInt Class by inheriting from the Int Class. The MInt Class has an integer field MOD = 107, and addition and multiplication return the remainder of the result divided by MOD, respectively.

Analyze the following IntApplication to understand the specifications of these classes.

```
class IntApplication{
   public static void main(String[] args){
      Int  a = new Int(0);
      Int  b = new MInt(0);
      MInt c = new MInt(0);

      c.setValue(1);

      System.out.println(a.getValue() + " (" + a.INT_MAX + ")");
      System.out.println(b.getValue() + " (" + b.INT_MAX + ")");
      System.out.println(c.getValue() + " (" + c.INT_MAX + ")");
      System.out.println("MOD = " + c.MOD);

      Int resA = a.add(new MInt(10)).add(new MInt(20)).mul(new MInt(30));
      Int resB = b.add(new MInt(10)).add(new MInt(20)).mul(new MInt(30));
      Int resC = c.mul(new MInt(10)).add(new MInt(20)).mul(new MInt(30));

      System.out.println(resA);
      System.out.println(resB);
      System.out.println(resC);
   }
}
```

Your implementation would be successful if you get the following output from IntApplication.

```
0 (1024)
0 (1024)
1 (1024)
MOD = 107
900
44
44
```

Submission Files	Types
D/Int.java	Java Class
D/MInt.java	Java Class
D/IntApplication.java	Java Class

Summary

In this exercise, we learned about the concept of inheritance, how to create a sub-class from a base-class as well as how to construct a class hierarchy. We studied some advantages of inheritance through a simple class hierarchy. Also, we checked the mechanism of the constructor chain and override.

Additionally, we have learned how to design applications by composing (delegating) classes. Inheritance is a powerful feature of object-oriented programming, but its adoption should be considered very carefully, because it creates tight relationships between classes, making them difficult to maintain. In general, composition should be your first consideration, and use of inheritance in the development of your application will be rare. There are several criteria for employing inheritance, including the following:

- The "is-a" relationship between Sub-class and Base-class must hold.
- The possibility of changes to the Base-class specification is low.
- The developer of Sub-class and Base-class is the same person, and he/she is familiar with the specification of Base-class.