Lab 7. Polymorphism & Abstraction

Writing Good Programs

The only way to learn programming is program, program and program. Learning programming is like learning cycling, swimming or any other sports. You can't learn by watching or reading books. Start to program immediately. On the other hands, to improve your programming, you need to read many books and study how the masters program.
It is easy to write programs that work. It is much harder to write programs that not only work but also easy to maintain and understood by others – I call these good programs. In the real world, writing program is not meaningful. You have to write good programs, so that others can understand and maintain your programs.
Pay particular attention to:

1. **Coding style**:
   - Read Java code convention: "Google Java Style Guide" or "Java Code Conventions - Oracle".
   - Follow the Java Naming Conventions for variables, methods, and classes STRICTLY. Use CamelCase for names. Variable and method names begin with lowercase, while class names begin with uppercase. Use nouns for variables (e.g., radius) and class names (e.g., Circle). Use verbs for methods (e.g., getArea(), isEmpty()).
   - **Use Meaningful Names**: Do not use names like a, b, c, d, x, x1, x2, and x1688 - they are meaningless. Avoid single-alphabet names like i, j, k. They are easy to type, but usually meaningless. Use single-alphabet names only when their meaning is clear, e.g., x, y, z for co-ordinates and i for array index. Use meaningful names like row and col (instead of x and y, i and j, x1 and x2), numStudents (not n), maxGrade, size (not n), and upperbound (not n again). Differentiate between singular and plural nouns (e.g., use books for an array of books, and book for each item).
   - Use consistent indentation and coding style. Many IDEs (such as Eclipse / NetBeans) can re-format your source codes with a single click.
2. **Program Documentation**: Comment! Comment! and more Comment to explain your code to other people and to yourself three days later.
3. The problems in this tutorial are certainly NOT challenging. There are tens of thousands of challenging problems available – used in training for various programming contests (such as International Collegiate Programming Contest (ICPC), International Olympiad in Informatics (IOI)).

1 Exercises on Polymorphism, Abstract Classes and Interfaces

1.1 Abstract Superclass Shape and Its Concrete Subclasses

Rewrite the superclass Shape and its subclasses Circle, Rectangle and Square, as shown in the class diagram.

Shape is an abstract class containing 2 abstract methods: *getArea()* and *getPerimeter()*, where its concrete subclasses must provide its implementation. All instance variables shall have protected access, i.e., accessible by its subclasses and classes in the same package. Mark all the overridden methods with annotation @Override.

In this exercise, Shape shall be defined as an abstract class, which contains:

- Two protected instance variables color(*String*) and filled (*boolean*). The protected variables can be accessed by its subclasses and classes in the same package. They are denoted with a '#' sign in the class diagram.
- Getter and setter for all the instance variables, and *toString()*.
- Two abstract methods *getArea()* and *getPerimeter()* (shown in italics in the class diagram).

The subclasses Circle and Rectangle shall override the abstract methods *getArea()* and *getPerimeter()* and provide the proper implementation. They also override the *toString()*. Write a test class to test these statements involving polymorphism and explain the outputs. Some statements may trigger compilation errors. Explain the errors, if any.

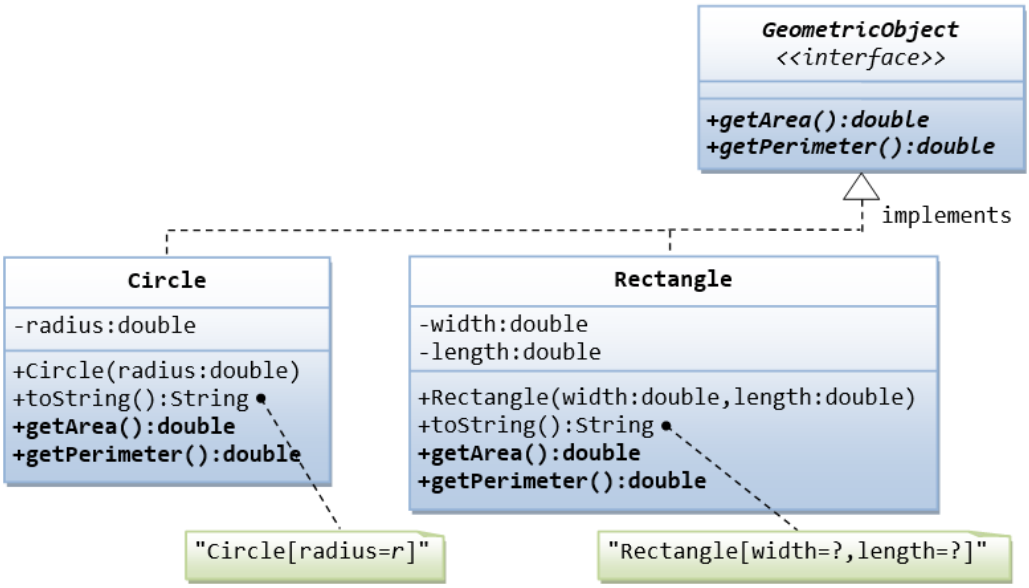
```
Shape shape1 = new Circle(5.5, "red", false);  // Upcast Circle to Shape
System.out.println(shape1);                    // which version?
System.out.println(shape1.getArea());          // which version?
System.out.println(shape1.getPerimeter());     // which version?
System.out.println(shape1.getColor());
System.out.println(shape1.isFilled());
System.out.println(shape1.getRadius());

Circle circle1 = (Circle)shape1;               // Downcast back to Circle
System.out.println(circle1);
System.out.println(circle1.getArea());
System.out.println(circle1.getPerimeter());
System.out.println(circle1.getColor());
System.out.println(circle1.isFilled());
System.out.println(circle1.getRadius());

Shape shape2 = new Shape();

Shape shape3 = new Rectangle(1.0, 2.0, "red", false);   // Upcast
System.out.println(shape3);
System.out.println(shape3.getArea());
System.out.println(shape3.getPerimeter());
System.out.println(shape3.getColor());
System.out.println(shape3.getLength());

Rectangle rectangle1 = (Rectangle)shape3;   // downcast
System.out.println(rectangle1);
System.out.println(rectangle1.getArea());
System.out.println(rectangle1.getColor());
System.out.println(rectangle1.getLength());

Shape shape4 = new Square(6.6);      // Upcast
System.out.println(shape4);
System.out.println(shape4.getArea());
System.out.println(shape4.getColor());
```

```
System.out.println(shape4.getSide());

// Take note that we downcast Shape shape4 to Rectangle,
//  which is a superclass of Square, instead of Square
Rectangle rectangle2 = (Rectangle)shape4;
System.out.println(rectangle2);
System.out.println(rectangle2.getArea());
System.out.println(rectangle2.getColor());
System.out.println(rectangle2.getSide());
System.out.println(rectangle2.getLength());

// Downcast Rectangle rectangle2 to Square
Square square1 = (Square)rectangle2;
System.out.println(square1);
System.out.println(square1.getArea());
System.out.println(square1.getColor());
System.out.println(square1.getSide());
System.out.println(square1.getLength());
```

What is the usage of the abstract method and abstract class?

1.2 GeometricObject Interface and its Implementation Classes Circle and Rectangle

Write an interface called GeometricObject, which contains 2 abstract methods: *getArea()* and *getPerimeter()*, as shown in the class diagram. Also write an implementation class called Circle. Mark all the overridden methods with annotation @Override.

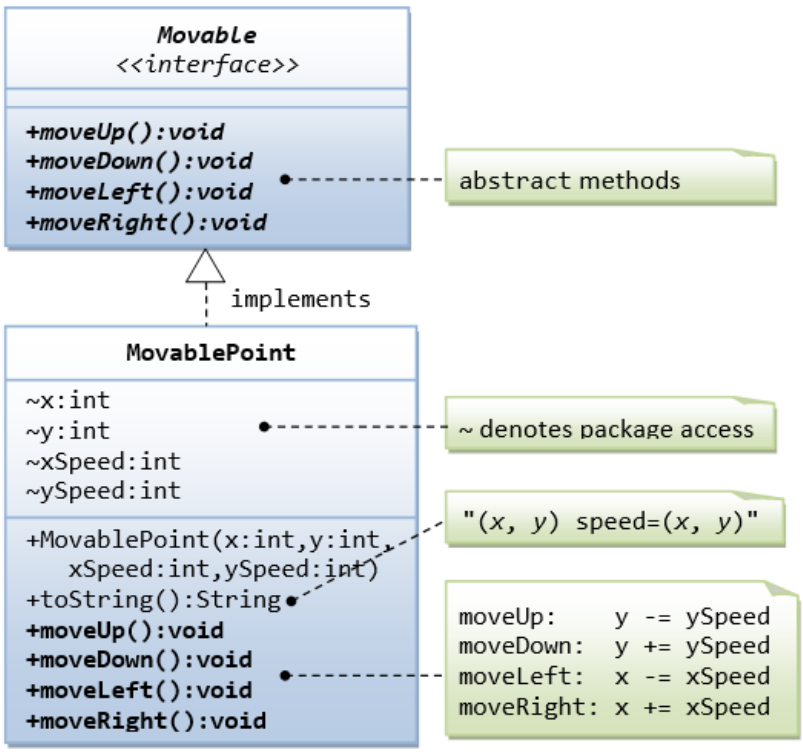


1.3 Movable Interface and its Implementation MovablePoint Class

Write an interface called Movable, which contains 4 abstract methods:

- *moveUp()*
- *moveDown()*
- *moveLeft()*
- *moveRight()*

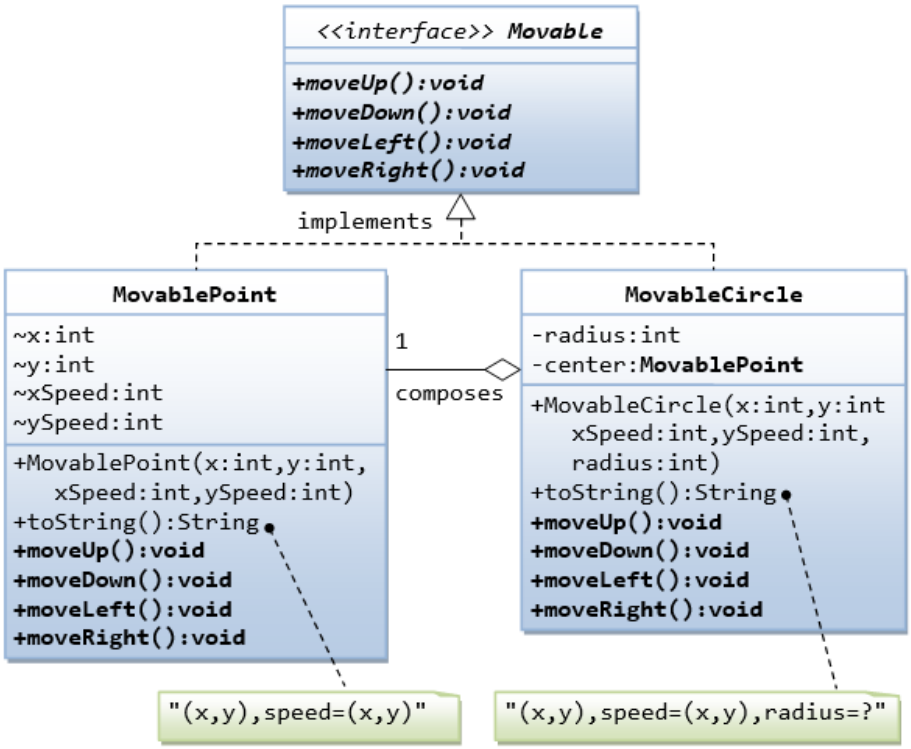
as shown in the class diagram.



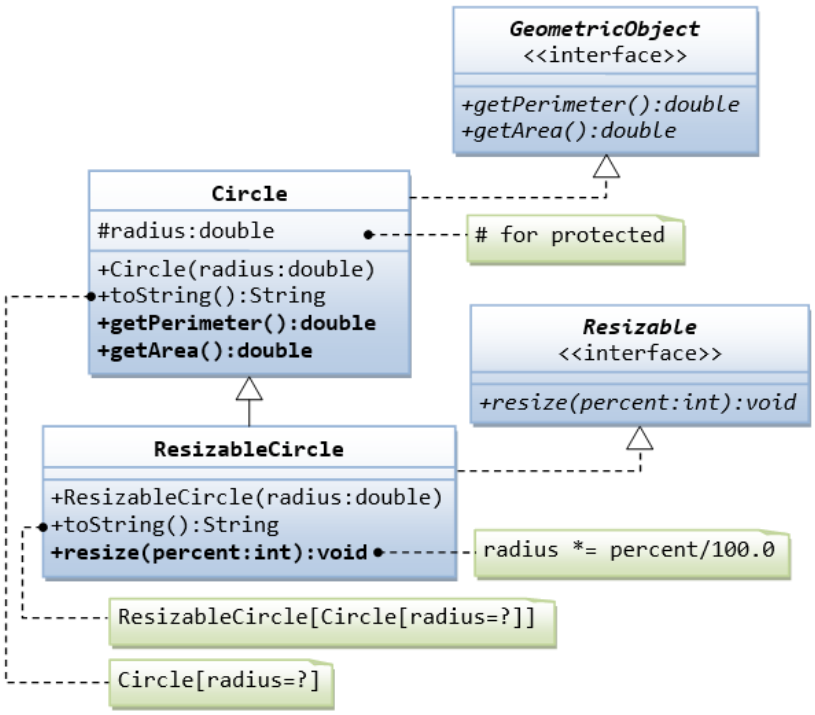
Also write an implementation class called MovablePoint. Mark all the overridden methods with annotation @Override.

1.4 Movable Interface and its Implementation Classes MovablePoint and MovableCircle

Write an interface called Movable, which contains 4 abstract methods *moveUp()*, *moveDown()*, *moveLeft()* and *moveRight()*, as shown in the class diagram. Also write the implementation classes called MovablePoint and MovableCircle. Mark all the overridden methods with annotation @Override.



1.5 Interfaces Resizable and GeometricObject



1. Write the interface called GeometricObject, which declares two abstract methods: *getParameter()* and *getArea()*, as specified in the class diagram.

   Hints:

```
public interface GeometricObject {
   public double getPerimeter();
   ......
}
```

2. Write the implementation class Circle, with a protected variable radius, which implements the interface GeometricObject.

   Hints:

```
public class Circle implements GeometricObject {
   // Private variable
   ......

   // Constructor
   ......

   // Implement methods defined in the interface GeometricObject
   @Override
   public double getPerimeter() { ..... }

   ......
}
```

3. Write a test program called TestCircle to test the methods defined in Circle.

4. The class ResizableCircle is defined as a subclass of the class Circle, which also implements an interface called Resizable, as shown in class diagram. The interface Resizable declares an abstract method *resize()*, which modifies the dimension (such as radius) by the given percentage. Write the interface Resizable and the class ResizableCircle.

   Hints:

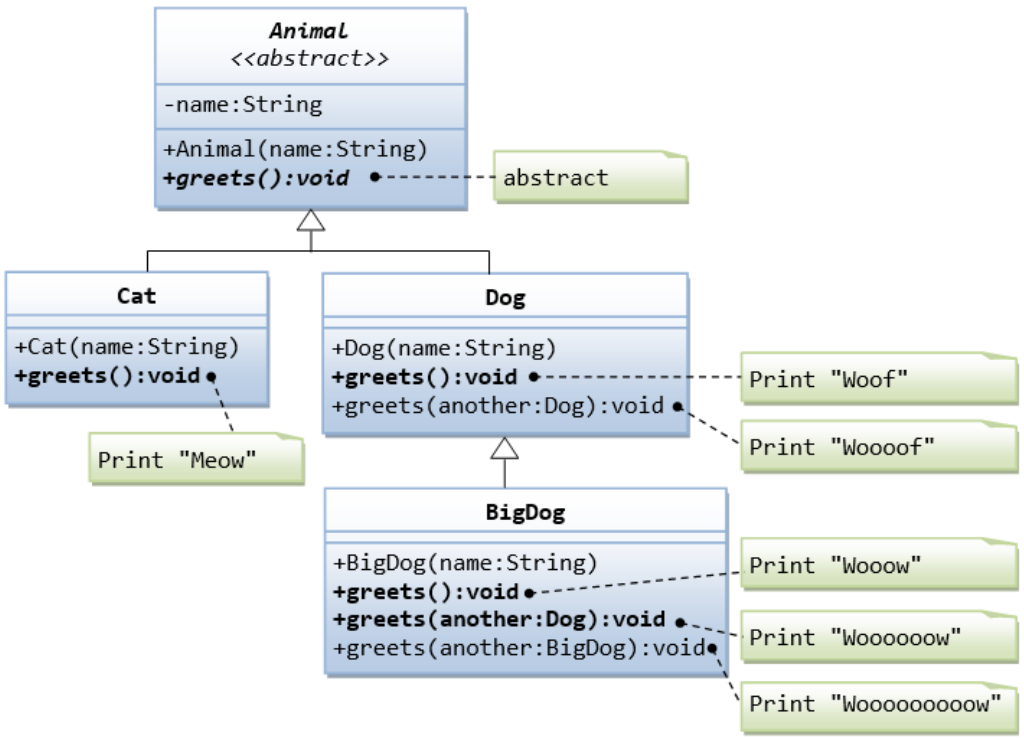
```
public interface Resizable {
   public double resize(...);
}
```

```java
public class ResizableCircle extends Circle implements Resizeable {

   // Constructor
   public ResizableCircle(double radius) {
      super(...);
   }

   // Implement methods defined in the interface Resizable
   @Override
   public double resize(int percent) { ..... }
}
```

5. Write a test program called TestResizableCircle to test the methods defined in ResizableCircle.

1.6 Abstract Superclass Animal and its Implementation Subclasses

Write the codes for all the classes shown in the class diagram. Mark all the overridden methods with annotation @Override.



1.7 Another View of Abstract Superclass Animal and its Implementation Subclasses

Examine the following codes and draw the class diagram.

```java
public abstract class Animal {
  abstract public void greeting();
}
```

```java
public class Cat extends Animal {
  @Override
  public void greeting() {
    System.out.println("Meow!");
  }
}
```

```java
public class Dog extends Animal {
  @Override
  public void greeting() {
    System.out.println("Woof!");
  }

  public void greeting(Dog another) {
    System.out.println("Wooooooooof!");
  }
}
```

```java
public class BigDog extends Dog {
  @Override
  public void greeting() {
    System.out.println("Woow!");
  }

  @Override
  public void greeting(Dog another) {
    System.out.println("Wooooowwwww!");
  }
}
```

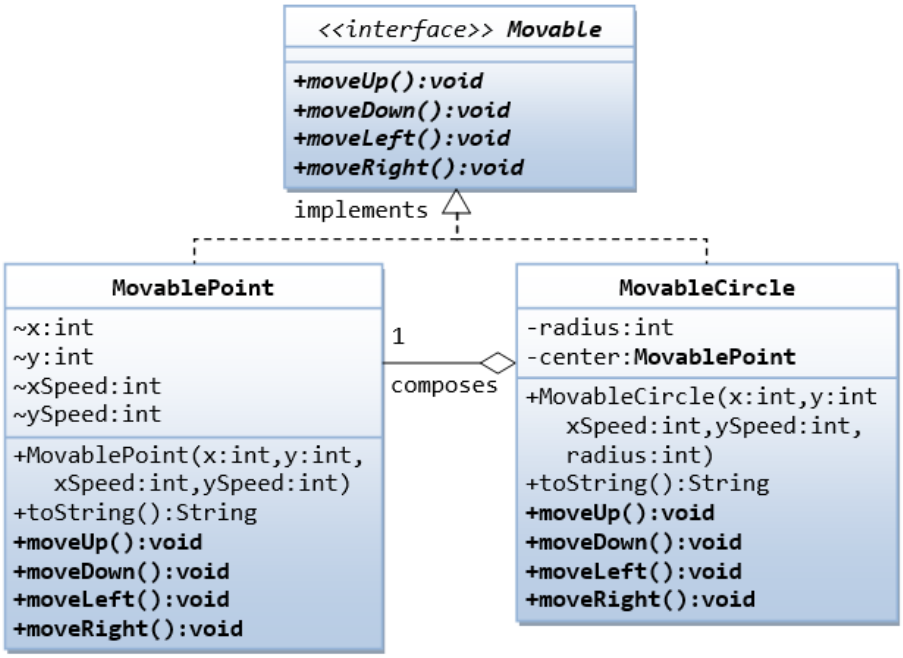
Explain the outputs (or error) for the following test program.

```
public class TestAnimal {
   public static void main(String[] args) {
      // Using the subclasses
      Cat cat1 = new Cat();
      cat1.greeting();
      Dog dog1 = new Dog();
      dog1.greeting();
      BigDog bigDog1 = new BigDog();
      bigDog1.greeting();

      // Using Polymorphism
      Animal animal1 = new Cat();
      animal1.greeting();
      Animal animal2 = new Dog();
      animal2.greeting();
      Animal animal3 = new BigDog();
      animal3.greeting();
      Animal animal4 = new Animal();

      // Downcast
      Dog dog2 = (Dog)animal2;
      BigDog bigDog2 = (BigDog)animal3;
      Dog dog3 = (Dog)animal3;
      Cat cat2 = (Cat)animal2;
      dog2.greeting(dog3);
      dog3.greeting(dog2);
      dog2.greeting(bigDog2);
      bigDog2.greeting(dog2);
      bigDog2.greeting(bigDog1);
   }
}
```

1.8 Interface Movable and its implementation subclasses MovablePoint and MovableCircle

Suppose that we have a set of objects with some common behaviors: they could move up, down, left or right. The exact behaviors (such as how to move and how far to move) depend on the objects themselves. One common way to model these common behaviors is to define an interface called Movable, with abstract methods *moveUp()*, *moveDown()*, *moveLeft()* and *moveRight()*. The classes that implement the Movable interface will provide actual implementation to these abstract methods.

Let's write two concrete classes - MovablePoint and MovableCircle - that implement the Movable interface.



The code for the interface Movable is straight forward.

```java
public interface Movable {  // saved as "Movable.java"
   public void moveUp();
   ......
}
```

For the MovablePoint class, declare the instance variable x, y, $xSpeed$ and $ySpeed$ with package access as shown with ' ' in the class diagram (i.e., classes in the same package can access these variables directly). For the MovableCircle class, use a MovablePoint to represent its center (which contains four variable x, y, $xSpeed$ and $ySpeed$). In other words, the MovableCircle composes a MovablePoint, and its radius.

```java
public class MovablePoint implements Movable { // saved as "MovablePoint.
   ↪ java"
   // instance variables
   int x;
   int y;
   int xSpeed;
   int ySpeed;     // package access

   // Constructor
   public MovablePoint(int x, int y, int xSpeed, int ySpeed) {
```

```
    this.x = x;
    ......
  }
  ......

  // Implement abstract methods declared in the interface Movable
  @Override
  public void moveUp() {
    y -= ySpeed;   // y-axis pointing down for 2D graphics
  }
  ......
}
```

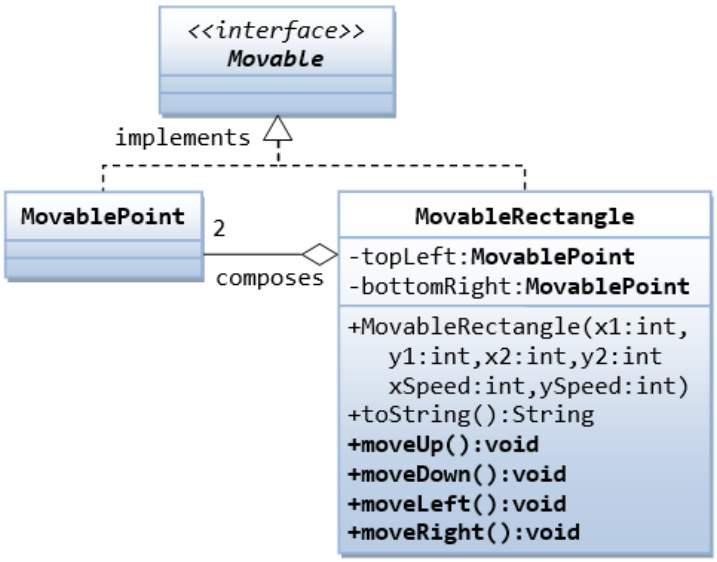
```
public class MovableCircle implements Movable { // saved as "
    ↪ MovableCircle.java"
  // instance variables
  private MovablePoint center;   // can use center.x, center.y directly
  //  because they are package accessible
  private int radius;

  // Constructor
  public MovableCircle(int x, int y, int xSpeed, int ySpeed, int radius)
      ↪ {
    // Call the MovablePoint's constructor to allocate the center instance.
    center = new MovablePoint(x, y, xSpeed, ySpeed);
    ......
  }
  ......

  // Implement abstract methods declared in the interface Movable
  @Override
  public void moveUp() {
    center.y -= center.ySpeed;
  }
  ......
}
```

Write a test program and try out these statements:

```
Movable m1 = new MovablePoint(5, 6, 10, 15);      // upcast
System.out.println(m1);
m1.moveLeft();
System.out.println(m1);
```

```
5
  Movable m2 = new MovableCircle(1, 2, 3, 4, 20);   // upcast
7 System.out.println(m2);
  m2.moveRight();
9 System.out.println(m2);
```

Write a new class called MovableRectangle, which composes two MovablePoints (representing the top-left and bottom-right corners) and implementing the Movable Interface. Make sure that the two points has the same speed.



What is the difference between an interface and an abstract class?