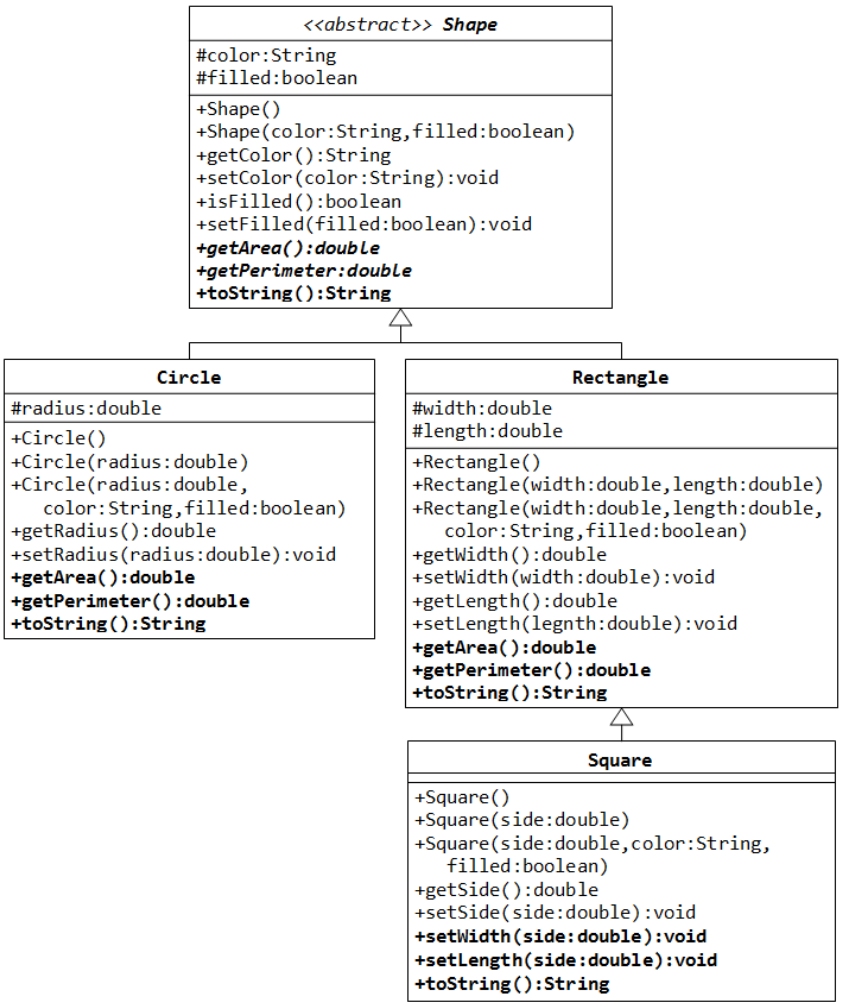
Exercise 10

Assignment 1: Abstract classes

This assignment is an extension of the shape example from exercise 7. The task is to Rewrite the superclass Shape and its subclasses Circle, Rectangle and Square, as shown in the class diagram.



Shape should be defined as an `abstract` class, which contains:

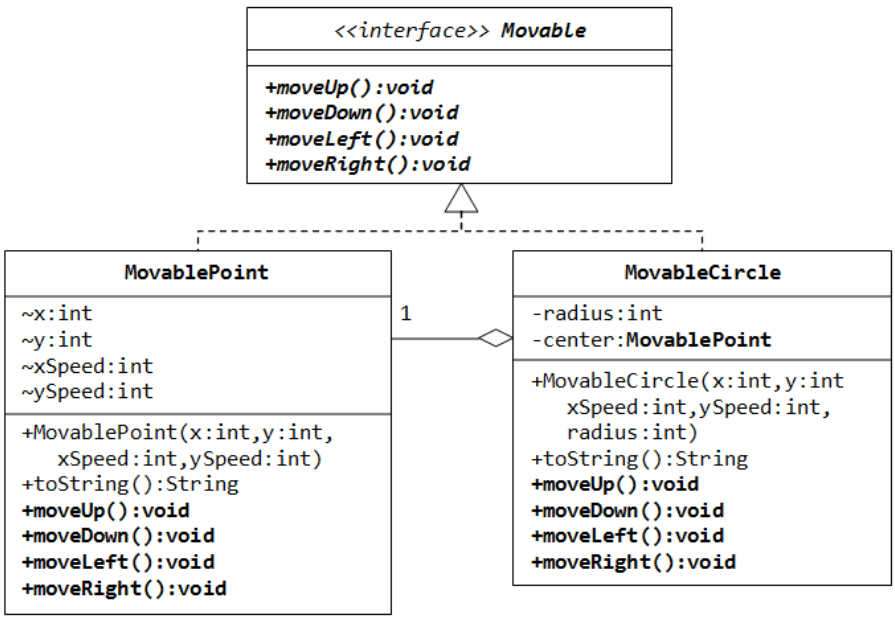
- Two `protected` instance variables `color`(`String`) and `filled`(`boolean`). The `protected` variables can be accessed by its subclasses and classes in the same package. They are denoted with a '`#`' sign in the class diagram.
- Getter and setter for all the instance variables, and `toString()`.
-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

The subclasses `Circle` and `Rectangle` should *override* the `abstract` methods `getArea()` and `getPerimeter()` and provide the proper implementation. They also *override* the `toString()`.
Overridden methods are bold in the class diagram.

Write a test class using the example code from below as inspiration to test these statements involving polymorphism and explain the outputs. Some statements may trigger compilation errors. Explain the errors, if any.

```
Shape s1 = new Circle(5.5, "RED", false);  // Upcast Circle to Shape
System.out.println(s1);                    // which version?
System.out.println(s1.getArea());          // which version?
System.out.println(s1.getPerimeter());     // which version?
System.out.println(s1.getColor());
System.out.println(s1.isFilled());
System.out.println(s1.getRadius());

Circle c1 = (Circle)s1;                   // Downcast back to Circle
System.out.println(c1);
System.out.println(c1.getArea());
System.out.println(c1.getPerimeter());
System.out.println(c1.getColor());
System.out.println(c1.isFilled());
System.out.println(c1.getRadius());

Shape s2 = new Shape();
Shape s3 = new Rectangle(1.0, 2.0, "RED", false);   // Upcast
System.out.println(s3);
System.out.println(s3.getArea());
System.out.println(s3.getPerimeter());
System.out.println(s3.getColor());
System.out.println(s3.getLength());

Rectangle r1 = (Rectangle)s3;   // downcast
System.out.println(r1);
System.out.println(r1.getArea());
System.out.println(r1.getColor());
System.out.println(r1.getLength());

Shape s4 = new Square(6.6);     // Upcast
System.out.println(s4);
System.out.println(s4.getArea());
System.out.println(s4.getColor());
System.out.println(s4.getSide());

// Take note that we downcast Shape s4 to Rectangle,
//  which is a superclass of Square, instead of Square
Rectangle r2 = (Rectangle)s4;
System.out.println(r2);
System.out.println(r2.getArea());
System.out.println(r2.getColor());
System.out.println(r2.getSide());
System.out.println(r2.getLength());

// Downcast Rectangle r2 to Square
Square sq1 = (Square)r2;
System.out.println(sq1);
System.out.println(sq1.getArea());
System.out.println(sq1.getColor());
System.out.println(sq1.getSide());
System.out.println(sq1.getLength());
```

Assignment 2: Interfaces



Suppose that we have a set of objects with some common behavior: they can move up, down, left or right. The exact behavior (such as how to move and how far to move) depends on the objects themselves.
To model these common behaviors we will define an *interface* called `Movable`, with the `abstract` methods `moveUp()`, `moveDown()`, `moveLeft()` and `moveRight()`. The classes that implement the `Movable` interface will provide actual implementation to these `abstract` methods.
First, write two concrete classes - `MovablePoint` and `MovableCircle` - that implement the Movable interface.

First we need the code for the interface.

```
public interface Movable {
   public void moveUp();
   // Add missing code
}
```

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). You can have look here for repeating the access modifiers (https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html).
Package access is achieved with not modifier (row 3 in the access levels table). 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`.

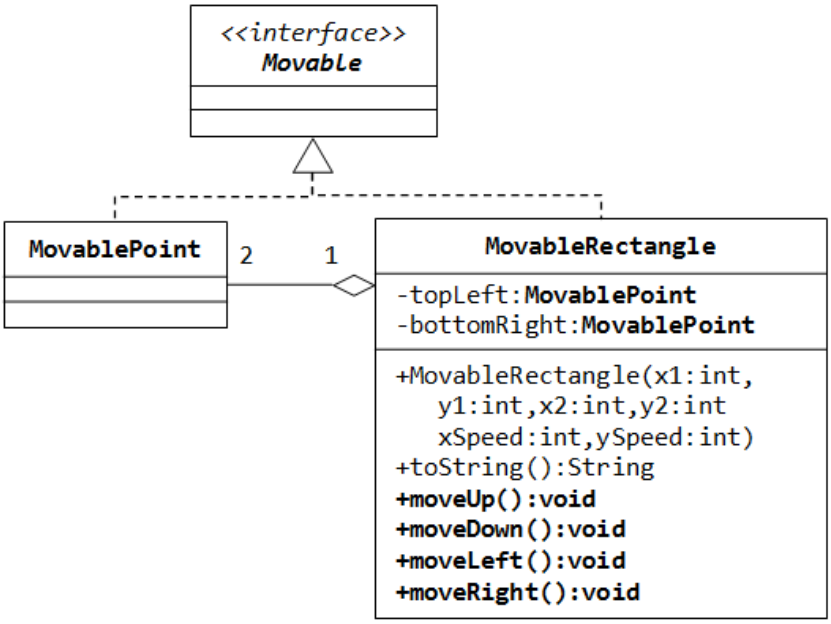
```
public class MovablePoint implements Movable { // saved as "MovablePoint.java"
   // instance variables
   int x, y, xSpeed, ySpeed;     // package access

   // Constructor
   public MovablePoint(int x, int y, int xSpeed, int ySpeed) {
      this.x = x;
```

```
      // Add code
   }

   // Implement abstract methods declared in the interface Movable
   @Override
   public void moveUp() {
      y += ySpeed;
   }
   // Add code
}
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.
      this.center = new MovablePoint(x, y, xSpeed, ySpeed);
      // Add code
   }

   // Implement abstract methods declared in the interface Movable
   @Override
   public void moveUp() {
      center.y += center.ySpeed;
   }
   // Add code missing methods and toString
}
```

Write a test class and try out these statements:

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

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

Write a new class called `MovableRectangle`, which composes (remember composition from previous exercises!) two `MovablePoints` (representing the top-left and bottom-right corners) and implementing the `Movable` Interface. See class diagram below. Make sure that the two points has the same speed.



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