# OOP Exercises

## Exercises on Defining Classes

Exercise (Circle): A class called **circle** is designed as shown in the following class diagram.

| Circle |
|---|
| -radius:double<br>-color:String |
| +Circle()<br>+Circle(radius:double)<br>+getRadius():double<br>+getArea():double |

It contains:

- Two `private` instance variables: `radius` (of the type `double`) and `color` (of the type `String`);
- Two *overloaded* constructors;
- Two public accessor methods: `getRadius()` and `getArea()`.

Compile "`Circle.java`". Can you run the `Circle` class? Why? This `Circle` class does not have a `main()` method. Hence, it cannot be run directly. This `Circle` class is a “building block” and is meant to be used in another program.

Try:

1. **(Constructor)** Modify the class `Circle` to include a third constructor for constructing a `Circle` instance with the given `radius` and `color`.
2. 
```
 // Construtor to construct a new instance of Circle with the given radius and color
public Circle (double r, String c) {......}
```

   Modify the test program `TestCircle` to construct an instance of `Circle` using this constructor.

3. **(Getter)** Add a getter for variable `color` for retrieving the `color` of a `Circle` instance.
4. 
```
 // Getter for instance variable color
public String getColor() {......}
```

   Modify the test program to try out this method.

5. **(public vs. private)** In `TestCircle`, can you access the instance variable `radius` directly (e.g., `System.out.println(c1.radius)`); or assign a new value to `radius` (e.g., `c1.radius=5.0`)? Try it out and explain the error messages.
6. **(Setter)** Is there a need to change the values of `radius` and `color` of a `Circle` instance after it is constructed? If so, add two `public` methods called *setters* for changing the `radius` and `color` of a `Circle` instance as follows:

```
7.  // Setter for instance variable radius
8.  public void setRadius(double r) {
9.     radius = r;
10. }
11.
12. // Setter for instance variable color
   public void setColor(String c) { ...... }
```

Modify the `TestCircle` to test these methods, e.g.,

```
Circle c3 = new Circle();   // construct an instance of Circle
c3.setRadius(5.0);          // change radius
c3.setColor(...);           // change color
```

13. **(Keyword "this")** Instead of using variable names such as `r` (for `radius`) and `c` (for `color`) in the methods' arguments, it is better to use `radius` (for `radius`) and `color` (for `color`) and use the special keyword "`this`" to resolve the conflict between instance variables and methods' arguments. For example,

```
14. // Instance variable
15. private double radius;
16.
17. // Setter of radius
18. public void setRadius(double radius) {
19.    this.radius = radius;   // "this.radius" refers to the 
   instance variable
20.                            // "radius" refers to the method's 
   argument
   }
```

Modify ALL the constructors and setters in the `Circle` class to use the keyword "this".

21. **(Method toString())** Every well-designed Java class should contain a `public` method called `toString()` that returns a short description of the instance (in a return type of `String`). The `toString()` method can be called explicitly (via *`instanceName`*`.toString()`) just like any other method; or implicitly through `println()`. If an instance is passed to the `println(`*`anInstance`*`)` method, the `toString()` method of that instance will be invoked implicitly. For example, include the following `toString()` methods to the `Circle` class:

```
22. public String toString() {
23.    return "Circle: radius=" + radius + " color=" + color;
   }
```

Try calling `toString()` method explicitly, just like any other method:

```
Circle c1 = new Circle(5.0);
System.out.println(c1.toString());   // explicit call
```

`toString()` is called implicitly when an instance is passed to `println()` method, for example,

```
Circle c2 = new Circle(1.2);
System.out.println(c2.toString());  // explicit call
System.out.println(c2);             // println() calls toString()
implicitly, same as above
System.out.println("Operator '+' invokes toString() too: " + c2);
```

**Exercise (Author and Book): A class called `Author` is designed as follows:**

| **Author** |
|---|
| -name:String<br>-email:String<br>-gender:char |
| +Author(name:String, email:String, gender:char)<br>+getName():String<br>+getEmail():String<br>+setEmail(email:String):void<br>+getGender():char<br>+toString():String |

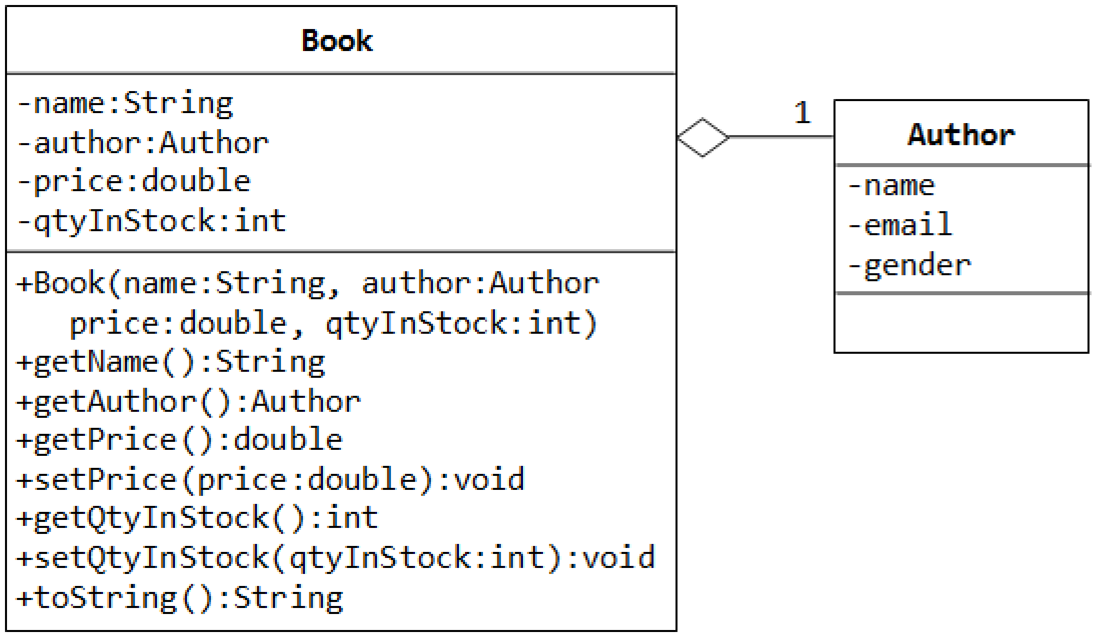
It contains:

- Three private instance variables: `name` (`String`), `email` (`String`), and `gender` (`char` of either `'m'` or `'f'`);
- One constructor to initialize the `name`, `email` and `gender` with the given values;
- Getters and setters: `getName()`, `getEmail()`, `setEmail()`, and `getGender()`. There is no setters for `name` and `gender` because these attributes cannot be changed.
- `toString()` that returns a description of this `Author` instance as "`name(email)`".

Write the `Author` class. Also write a test program called `TestAuthor` to test the constructor and public methods. Try changing the `email` of an author, e.g.,

```
Author anAuthor = new Author("Tan Ah Teck", "ahteck@somewhere.com",
'm');
System.out.println(anAuthor);   // call toString()
anAuthor.setEmail("paul@nowhere.com")
System.out.println(anAuthor);
```

A class called `Book` is designed as follows:




It contains:

- Four `private` instance variables: `name` (`String`), `author` (of the class `Author` you have just created), `price` (`double`), and `qtyInStock` (`int`). Assuming that each book is written by one author.
- One constructor which constructs an instance with the values given.
- Getters and setters: `getName()`, `getAuthor()`, `getPrice()`, `setPrice()`, `getQtyInStock()`, `setQtyInStock()`. Again there is no setter for `name` and `author`.
- `toString()` that return a description of this `Book` instance as "`"bookname" written by authorname(email)`". You should use `Author`'s `toString()` in this method.

Write the class `Book` (which uses the `Author` class written earlier). Also write a test program called `TestBook` to test the constructor and public methods in the class `Book`. Take Note that you have to construct an instance of `Author` before you can construct an instance of `Book`. E.g.,

```
Author anAuthor = new Author(......);
Book aBook = new Book("Java for dummy", anAuthor, 19.95, 1000);
// Use an anonymous instance of Author
Book anotherBook = new Book("more Java for dummy", new Author(......),
29.95, 888);
```

Take note that both `Book` and `Author` classes have a variable called `name`. However, it can be differentiated via the referencing instance. For a `Book` instance says `aBook`,

`aBook.name` refers to the `name` of the book; whereas for an `Author`'s instance say `auAuthor`, `anAuthor.name` refers to the `name` of the author. There is no need (and not recommended) to call the variables `bookName` and `authorName`.

Try:

1. Printing the `name` and `email` of the author from a `Book` instance. (Hint: `aBook.getAuthor().getName()`, `aBook.getAuthor().getEmail()`).
2. Introduce new methods called `getAuthorName()`, `getAuthorEmail()`, `getAuthorGender()` in the `Book` class to return the `name`, `email` and `gender` of the author of the book. For example,

   ```
   public String getAuthorName() { ...... }
   ```

Exercise (MyPoint): A class called `MyPoint`, which models a 2D point with `x` and `y` coordinates, is designed as follows:

| **MyPoint** |
|---|
| -x:int<br>-y:int |
| +MyPoint()<br>+MyPoint(x:int, y:int)<br>+getX():int<br>+setX(x:int):void<br>+getY():int<br>+setY(y:int):void<br>+setXY(x:int, y:int):void<br>+toString():String<br>+distance(x:int, y:int):double<br>+distance(another:MyPoint):double |

The class contains:

- Two instance variables `x`(`int`) and `y`(`int`).
- A "no-argument" or "no-arg" constructor that construct a point at `(0, 0)`.
- A constructor that constructs a point with the given `x` and `y` coordinates.
- Getter and setter for the instance variables `x` and `y`.
- A method `setXY()` to set both `x` and `y`.
- A `toString()` method that returns a string description of the instance in the format "`(x, y)`".
- A method called `distance(int x, int y)` that returns the distance from *this* point to another point at the given `(x, y)` coordinates.

- An overloaded `distance(MyPoint another)` that returns the distance from *this* point to the given `MyPoint` instance `another`.

Write a test program to test all the methods defined in the class.

Hints:

```
// Overloading method distance()
public double distance(int x, int y) {   // this version takes two ints
as arguments
   int xDiff = this.x - x;
   int yDiff = ......
   return Math.sqrt(xDiff*xDiff + yDiff*yDiff);
}

public double distance(MyPoint another) { // this version takes a
MyPoint instance as argument
   int xDiff = this.x - another.x;
   .......
}

// Test program
MyPoint p1 = new MyPoint(3, 0);
MyPoint p2 = new MyPoint(0, 4);
......
// Testing the overloaded method distance()
System.out.println(p1.distance(p2));    // which version?
System.out.println(p1.distance(5, 6));  // which version?
.....
```

Write a program that allocates `10` points in an array of `MyPoint`, and initializes to `(1, 1)`, `(2, 2)`, ... `(10, 10)`.

Hints: You need to allocate the array, as well as each of the ten `MyPoint` instances.

```
MyPoint[] points = new MyPoint[10]; // Declare and allocate an array of
MyPoint
for (......) {
   points[i] = new MyPoint(...);    // Allocate each of MyPoint
instances
}
```

Exercise (MyComplex): A class called `MyComplex`, which models complex numbers `x+yi`, is designed as follow:

| MyComplex |
|---|
| -real:double<br>-imag:double |
| +MyComplex(real:double, imag:double)<br>+getReal():double<br>+setReal(real:double):void<br>+getImag():double<br>+setImag(imag:double):void<br>+setValue(real:double, imag:double):void<br>+toString():String<br>+isReal():boolean<br>+isImaginary():boolean<br>+equals(real:double, imag:double):boolean<br>+equals(another:MyComplex):boolean<br>+magnitude():double<br>+argumentInRadians():double<br>+argumentInDegrees():int<br>+conjugate():MyComplex<br>+add(another:MyComplex):MyComplex<br>+subtract(another:MyComplex):MyComplex<br>+multiplyWith(another:MyComplex):MyComplex<br>+divideBy(another:MyComplex):MyComplex |

The class contains:

- Two instance variable named `real`(`double`) and `imag`(`double`) which stores the real and imaginary parts of the complex number respectively.
- A constructor that creates a `MyComplex` instance with the given real and imaginary values.
- Getters and setters for instance variables `real` and `imag`.
- A method `setValue()` to set the value of the complex number.
- A `toString()` that returns "`(x + yi)`" where `x` and `y` are the real and imaginary parts respectively.
- Methods `isReal()` and `isImaginary()` that returns `true` if this complex number is real or imaginary, respectively. Hint:

  ```
  return (imag == 0);   // isReal()
  ```

- A method `equals(double real, double imag)` that returns `true` if *this* complex number is equal to the given complex number of (real, imag).
- An overloaded `equals(MyComplex another)` that returns `true` if *this* complex number is equal to the given `MyComplex` instance `another`.

- A method magnitude()that returns the magnitude of this complex number.

  ```
  magnitude(x+yi) = Math.sqrt(x2 + y2)
  ```

- Methods `argumentInRadians()` and `argumentInDegrees()` that returns the argument of this complex number in radians (in `double`) and degrees (in `int`) respectively.

  ```
  arg(x+yi) = Math.atan2(y, x) (in radians)
  ```

  Note: The `Math` library has two arc-tangent methods, `Math.atan(double)` and `Math.atan2(double, double)`. We commonly use the `Math.atan2(y, x)` instead of `Math.atan(y/x)` to avoid division by zero. Read the documentation of `Math` class in package `java.lang`.

- A method `conjugate()` that returns a new `MyComplex` instance containing the complex conjugate of this instance.

  ```
  conjugate(x+yi) = x - yi
  ```

  Hint:

  ```
  return new MyComplex(real, -imag);  // construct a new instance
  and return the constructed instance
  ```

- Methods `add(MyComplex another)` and `subtract(MyComplex another)` that adds and subtract this instance with the given `MyComplex` instance `another`, and returns a new `MyComplex` instance containing the result.
- ```
  (a + bi) + (c + di) = (a+c) + (b+d)i
  (a + bi) - (c + di) = (a-c) + (b-d)i
  ```

- Methods `multiplyWith(MyComplex another)` and `divideBy(MyComplex another)` that multiplies and divides this instance with the given `MyComplex` instance `another`, keep the result in this instance, and returns this instance.
- ```
  (a + bi) * (c + di) = (ac - bd) + (ad + bc)i
  (a + bi) / (c + di) = [(a + bi) * (c – di)] / (c2 + d2)
  ```

  Hint:

  ```
  return this;  // return "this" instance
  ```

You are required to:

1. Write the `MyComplex` class.
2. Write a test program to test all the methods defined in the class.

3. Write an application called `MyComplexApp` that uses the `MyComplex` class. The application shall prompt the user for two complex numbers, print their values, check for real, imaginary and equality, and carry out all the arithmetic operations.

```
Enter complex number 1 (real and imaginary part): 1.1 2.2
Enter complex number 2 (real and imaginary part): 3.3 4.4

Number 1 is: (1.1 + 2.2i)
(1.1 + 2.2i) is NOT a pure real number
(1.1 + 2.2i) is NOT a pure imaginary number

Number 2 is: (3.3 + 4.4i)
(3.3 + 4.4i) is NOT a pure real number
(3.3 + 4.4i) is NOT a pure imaginary number

(1.1 + 2.2i) is NOT equal to (3.3 + 4.4i)
(1.1 + 2.2i) + (3.3 + 4.4i) = (4.4 + 6.6000000000000005i)
(1.1 + 2.2i) - (3.3 + 4.4i) = (-2.1999999999999997 + -2.2i)
```

Take note that there are a few flaws in the design of this class, which was introduced solely for teaching purpose:

- Comparing doubles in `equal()` using "==" may produce unexpected outcome. For example, `(2.2+4.4)==6.6` returns `false`. It is common to define a small threshold called `EPSILON` (set to about `10^-8`) for comparing floating point numbers.
- The method `add()`, `subtract()`, and `conjugate()` produce new instances, whereas `multiplyWith()` and `divideBy()` modify this instance. There is inconsistency in the design (introduced for teaching purpose).
- Unusual to have both `argumentInRadians()` and `argumentInDegrees()`.

Exercise (MyPolynomial): A class called `MyPolynomial`, which models polynomials of degree-n, is designed as shown:

$$c_n x^n + c_{n-1} x^{n-1} + \cdots + c_1 x + c_0.$$

| **MyPolynomial** |
|---|
| -coeffs:double[] |
| +MyPolynomial(coeffs:double...)<br>+MyPolynomial(filename:String)<br>+getDegree():int<br>+toString():String<br>+evaluate(x:double):double<br>+add(another:MyPolynomial):MyPolynomial<br>+multiply(another:MyPolynomial):MyPolynomial |

The class contains:

- An instance variable named `coeffs`, which stores the coefficients of the n-degree polynomial in a `double` array of size `n+1`, where $c_0$ is kept at index 0.
- A constructor `MyPolynomial(coeffs:double...)` that takes a variable number of doubles to initialize the coeffs array, where the first argument corresponds to $c_0$. The three dots is known as *varargs* (variable number of arguments), which is a new feature introduced in JDK 1.5. It accepts an array or a sequence of comma-separated arguments. The compiler automatically packs the comma-separated arguments in an array. The three dots can only be used for the last argument of the method. Hints:

```
  public class MyPolynomial {
     private double[] coeffs;
     public MyPolynomial(double... coeffs) {  // varargs
        this.coeffs = coeffs;                 // varargs is treated as array
     }
     ......
  }

  // Test program
  // Can invoke with a variable number of arguments
  MyPolynomial p1 = new MyPolynomial(1.1, 2.2, 3.3);
  MyPolynomial p1 = new MyPolynomial(1.1, 2.2, 3.3, 4.4, 5.5);
  // Can also invoke with an array
  Double coeffs = {1.2, 3.4, 5.6, 7.8}
MyPolynomial p2 = new MyPolynomial(coeffs);
```

- Another constructor that takes coefficients from a file (of the given `filename`), having this format:

```
  Degree-n(int)
  c0(double)
  c1(double)
  ......
  ......
  cn-1(double)
  cn(double)
  (end-of-file)
```

Hints:

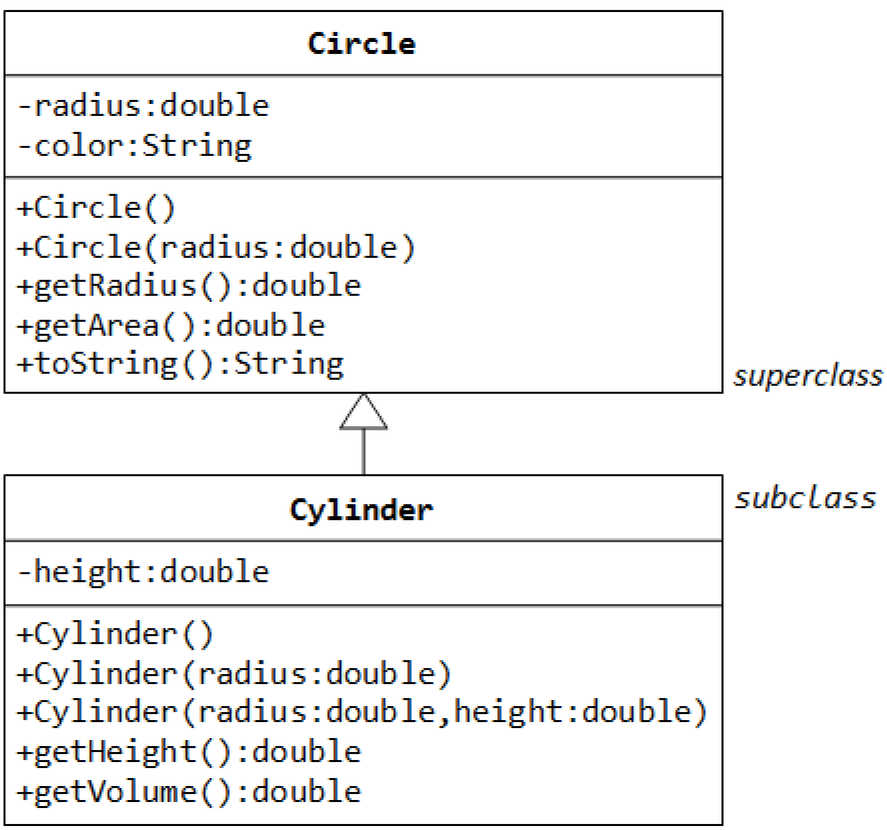
```
public MyPolynomial(String filename) {
   Scanner in = null;
   try {
      in = new Scanner(new File(filename));  // open file
   } catch (FileNotFoundException e) {
      e.printStackTrace();
   }
   int degree = in.nextInt();      // read the degree
```

```
        coeffs = new double[degree+1];  // allocate the array
        for (int i=0; i<coeffs.length; i++) {
           coeffs[i] = in.nextDouble();
        }
     }
```

- A method `getDegree()` that returns the degree of this polynomial.
- A method `toString()` that returns "$c_n$x^n+$c_{n-1}$x^(n-1)+...+$c_1$x+$c_0$".
- A method `evaluate(double x)` that evaluate the polynomial for the given `x`, by substituting the given `x` into the polynomial expression.
- Methods `add()` and `multiply()` that adds and multiplies this polynomial with the given `MyPolynomial` instance `another`, and returns a new `MyPolynomial` instance that contains the result.

## Exercises on Inheritance

Exercise (Circle and Cylinder): In this exercise, a subclass called `Cylinder` is derived from the superclass `Circle` as shown in the following class diagram (an arrow pointing up from the subclass to its superclass). Study how the subclass `Cylinder` invokes the superclass' constructors (`super()`, `super(radius)`) and inherits the variables and methods from the superclass `Circle`.



You can reuse the `Circle` class that you have created in the previous exercise. Make sure that you keep "`Circle.class`" in the same directory.

Write a test program (says `TestCylinder`) to test the `Cylinder` class created, as follow:

**Method Overriding and "Super"**: The subclass `Cylinder` inherits `getArea()` method from its superclass Circle. Try *overriding* the `getArea()` method in the subclass `Cylinder` to compute the surface area (=2π×radius×height + 2×base-area) of the cylinder instead of base area. That is, if `getArea()` is called by a `Circle` instance, it returns the area. If `getArea()` is called by a `Cylinder` instance, it returns the surface area of the cylinder.

If you override the `getArea()` in the subclass `Cylinder`, the `getVolume()` no longer works. This is because the `getVolume()` uses the *overridden* `getArea()` method found in the same class. (Java runtime will search the superclass only if it cannot locate the method in this class). Fix the `getVolume()`.

Hints: After overridding the `getArea()` in subclass `Cylinder`, you can choose to invoke the `getArea()` of the superclass `Circle` by calling `super.getArea()`.

Try:

Provide a `toString()` method to the `Cylinder` class, which overrides the `toString()` inherited from the superclass `Circle`, e.g.,

```
@Override
public String toString() {      // in Cylinder class
   return "Cylinder: subclass of " + super.toString()  // use Circle's
toString()
          + " height=" + height;
}
```

Try out the `toString()` method in `TestCylinder`.

Note: `@Override` is known as *annotation* (introduced in JDK 1.5), which asks compiler to check whether there is such a method in the superclass to be overridden. This helps greatly if you misspell the name of the `toString()`. If `@Override` is not used and `toString()` is misspelled as `ToString()`, it will be treated as a new method in the subclass, instead of overriding the superclass. If `@Override` is used, the compiler will signal an error. `@Override` annotation is optional, but certainly nice to have.

Exercise (Shape and subclasses Circle, Rectangle and Square): Write a superclass called `Shape` (as shown in the class diagram), which contains:

- Two instance variables `color`(`String`) and `filled`(`boolean`).

- Two constructors: a no-arg (no-argument) constructor that initializes the `color` to "green" and `filled` to `true`, and a constructor that initializes the `color` and `filled` to the given values.
- Getter and setter for all the instance variables. By convention, the getter for a `boolean` variable `xxx` is called `isXXX()` (instead of `getXxx()` for all the other types).
- A `toString()` method that returns "`A Shape with color of xxx and filled/Not filled`".

Write a test program to test all the methods defined in `Shape`.

Write two subclasses of `Shape` called `Circle` and `Rectangle`, as shown in the class diagram.

The Circle class contains:

- An instance variable `radius`(`double`).
- Three constructors as shown. The no-arg constructor initializes the radius to `1.0`.
- Getter and setter for the instance variable `radius`.
- Methods `getArea()` and `getPerimeter()`.
- Override the `toString()` method inherited, to return "`A Circle with radius=xxx, which is a subclass of yyy`", where `yyy` is the output of the `toString()` method from the superclass.

The Rectangle class contains:

- Two instance variables `width`(`double`) and `length`(`double`).
- Three constructors as shown. The no-arg constructor initializes the `width` and `length` to `1.0`.
- Getter and setter for all the instance variables.
- Methods `getArea()` and `getPerimeter()`.
- Override the `toString()` method inherited, to return "`A Rectangle with width=xxx and length=zzz, which is a subclass of yyy`", where `yyy` is the output of the `toString()` method from the superclass.

Write a class called `Square`, as a subclass of `Rectangle`. Convince yourself that `Square` can be modeled as a subclass of `Rectangle`. `Square` has no instance variable, but inherits the instance variables width and length from its superclass Rectangle.

- Provide the appropriate constructors (as shown in the class diagram). Hint:
- `public Square(double side) {`
- `super(side, side);  // Call superclass Rectangle(double, double)`
  `}`

- Override the `toString()` method to return "`A Square with side=xxx, which is a subclass of yyy`", where `yyy` is the output of the `toString()` method from the superclass.
- Do you need to override the `getArea()` and `getPerimeter()`? Try them out.
- Override the `setLength()` and `setWidth()` to change both the `width` and `length`, so as to maintain the square geometry.

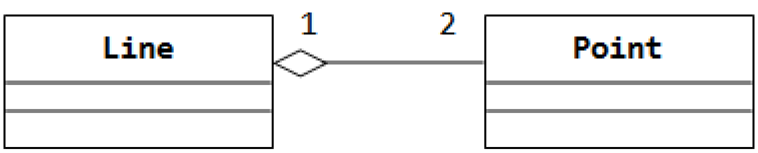
## Exercises on Composition vs Inheritance

They are two ways to reuse a class in your applications: *composition* and *inheritance*.

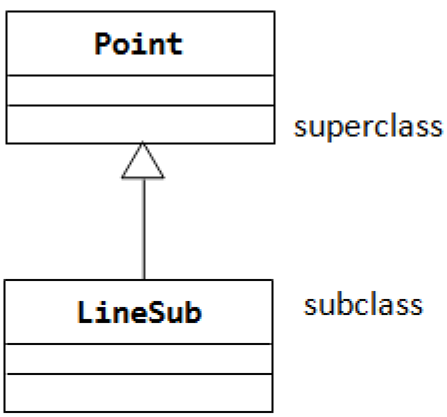
Exercise (Point and Line): Let us begin with *composition* with the statement "a line composes of two points".

Complete the definition of the following two classes: `Point` and `Line`. The class `Line` composes 2 instances of class `Point`, representing the beginning and ending points of the line. Also write test classes for `Point` and `Line` (says `TestPoint` and `TestLine`).

The class diagram for *composition* is as follows (where a diamond-hollow-head arrow pointing to its constituents):




Instead of *composition*, we can design a `Line` class using `inheritance`. Instead of "a line composes of two points", we can say that "a line is a point extended by another point", as shown in the following class diagram:

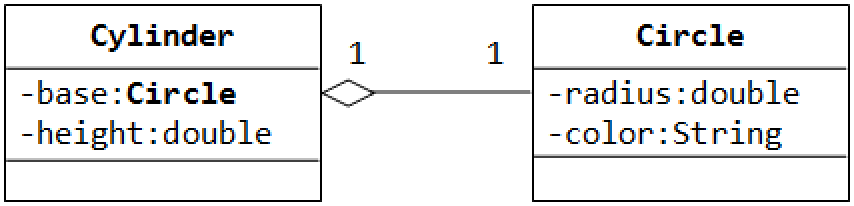



Let's re-design the `Line` class (called `LineSub`) as a subclass of class `Point`. `LineSub` inherits the starting point from its superclass `Point`, and adds an ending point. Complete the class definition. Write a testing class called `TestLineSub` to test `LineSub`.

```
public class LineSub extends Point {
   // A line needs two points: begin and end.
   // The begin point is inherited from its superclass Point.
   // Private variables
   Point end;               // Ending point

   // Constructors
   public LineSub (int beginX, int beginY, int endX, int endY) {
      super(beginX, beginY);             // construct the begin Point
      this.end = new Point(endX, endY);  // construct the end Point
   }
   public LineSub (Point begin, Point end) {  // caller to construct
the Points
      super(begin.getX(), begin.getY());      // need to reconstruct
the begin Point
      this.end = end;
   }

   // Public methods
   // Inherits methods getX() and getY() from superclass Point
   public String toString() { ... }

   public Point getBegin() { ... }
   public Point getEnd() { ... }
   public void setBegin(...) { ... }
   public void setEnd(...) { ... }

   public int getBeginX() { ... }
   public int getBeginY() { ... }
   public int getEndX() { ... }
   public int getEndY() { ... }

   public void setBeginX(...) { ... }
   public void setBeginY(...) { ... }
   public void setBeginXY(...) { ... }
   public void setEndX(...) { ... }
   public void setEndY(...) { ... }
   public void setEndXY(...) { ... }

   public int getLength() { ... }       // Length of the line
   public double getGradient() { ... }  // Gradient in radians
}
```

Summary: There are two approaches that you can design a line, `composition` or `inheritance`. "A line composes two points" or "A line is a point extended with another point"". Compare the `Line` and `LineSub` designs: `Line` uses *composition* and `LineSub` uses *inheritance*. Which design is better?

Exercise (Circle and Cylinder using composition): Try rewriting the Circle-Cylinder of the previous exercise using `composition` (as shown in the following class diagram) instead of `inheritance`. That is, "a cylinder is composed of a base circle and a height".

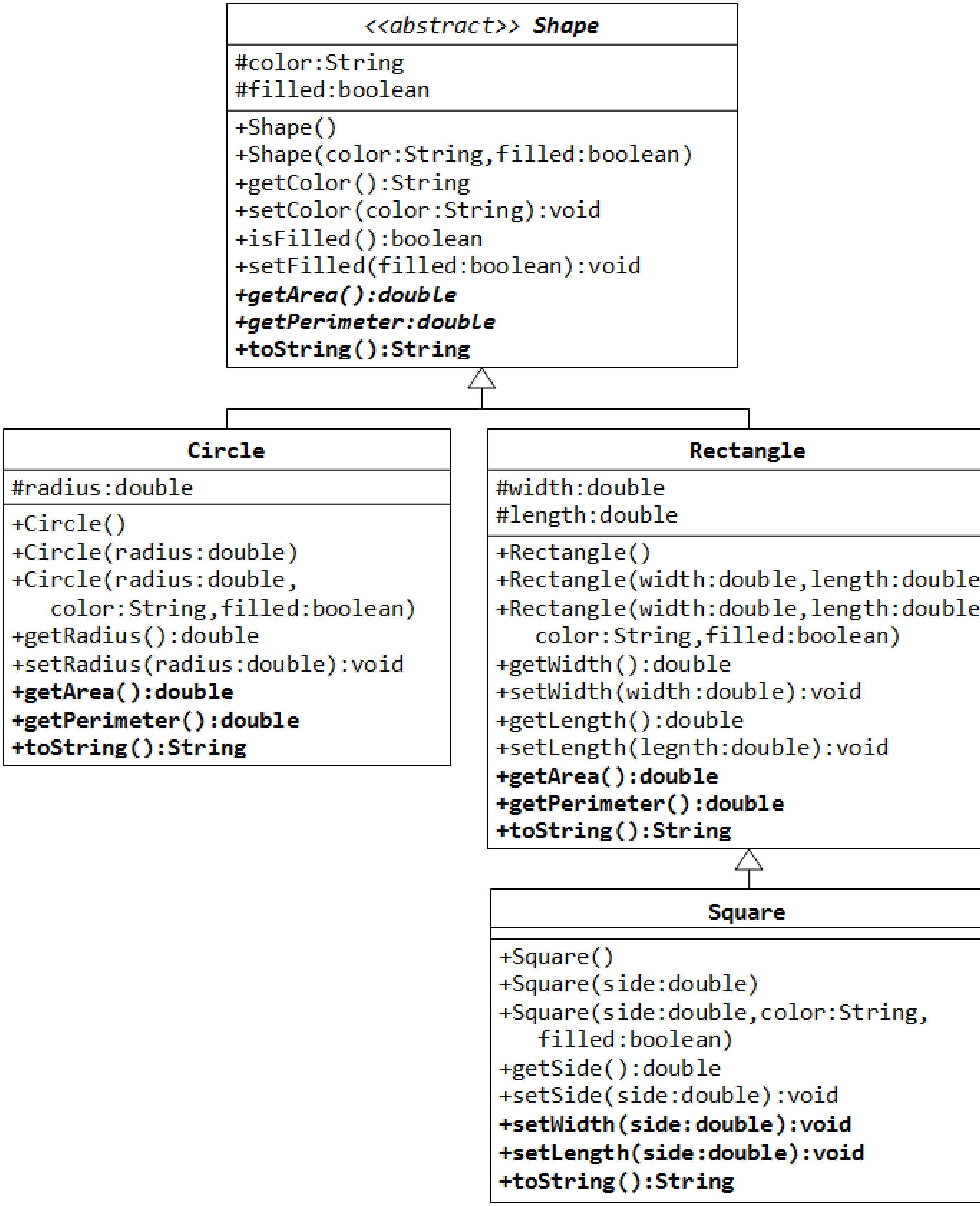



```
public class Cylinder {
   private Circle base;   // Base circle, an instance of Circle class
   private double height;

   // Constructor with default color, radius and height
   public Cylinder() {
      base = new Circle(); // Call the constructor to construct the
Circle
      height = 1.0;
   }
   ......
}
```

Which design (inheritance or composition) is better?

## Exercises on Polymorphism, Abstract Classes and Interfaces

Exercise (Abstract superclass Shape and its concrete subclasses): Rewrite the superclass `Shape` and its subclasses `Circle`, `Rectangle` and `Square`, as shown in the class diagram.



<<abstract>> Shape
#color:String
#filled:boolean
+Shape()
+Shape(color:String,filled:boolean)
+getColor():String
+setColor(color:String):void
+isFilled():boolean
+setFilled(filled:boolean):void
+getArea():double
+getPerimeter:double
+toString():String
Circle
#radius:double
+Circle()
+Circle(radius:double)
+Circle(radius:double,
color:String,filled:boolean)
+getRadius():double
+setRadius(radius:double):void
+getArea():double
+getPerimeter():double
+toString():String
Rectangle
#width:double
#length:double
+Rectangle()
+Rectangle(width:double,length:double
+Rectangle(width:double,length:double
color:String,filled:boolean)
+getWidth():double
+setWidth(width:double):void
+getLength():double
+setLength(legnth:double):void
+getArea():double
+getPerimeter():double
+toString():String
Square
+Square()
+Square(side:double)
+Square(side:double,color:String,
filled:boolean)
+getSide():double
+setSide(side:double):void
+setWidth(side:double):void
+setLength(side:double):void
+toString():String

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 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)sq1;
System.out.println(sq1);
System.out.println(sq1.getArea());
System.out.println(sq1.getColor());
System.out.println(sq1.getSide());
System.out.println(sq1.getLength());
```

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

Exercise (Polymorphism): Examine the following codes and draw the class diagram.

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

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

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

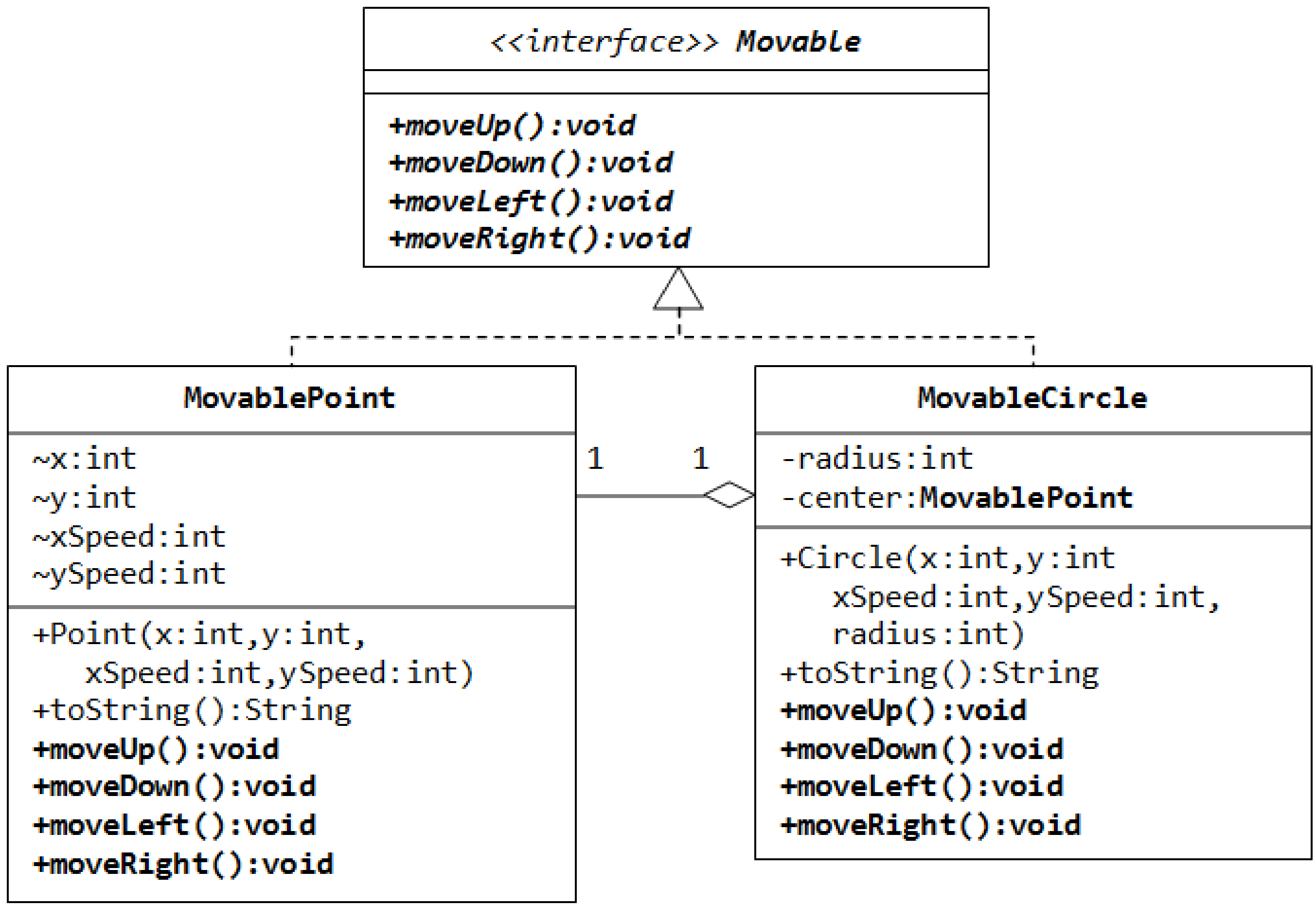
Explain the outputs (or error) for the following test program.

```
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);
   }
}
```

Exercise (Interface Movable and its implementations): Suppose that we have a set of objects with some common behaviors. They could move up, down, left or right. The exact behaviors (e.g., 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.

```
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`.

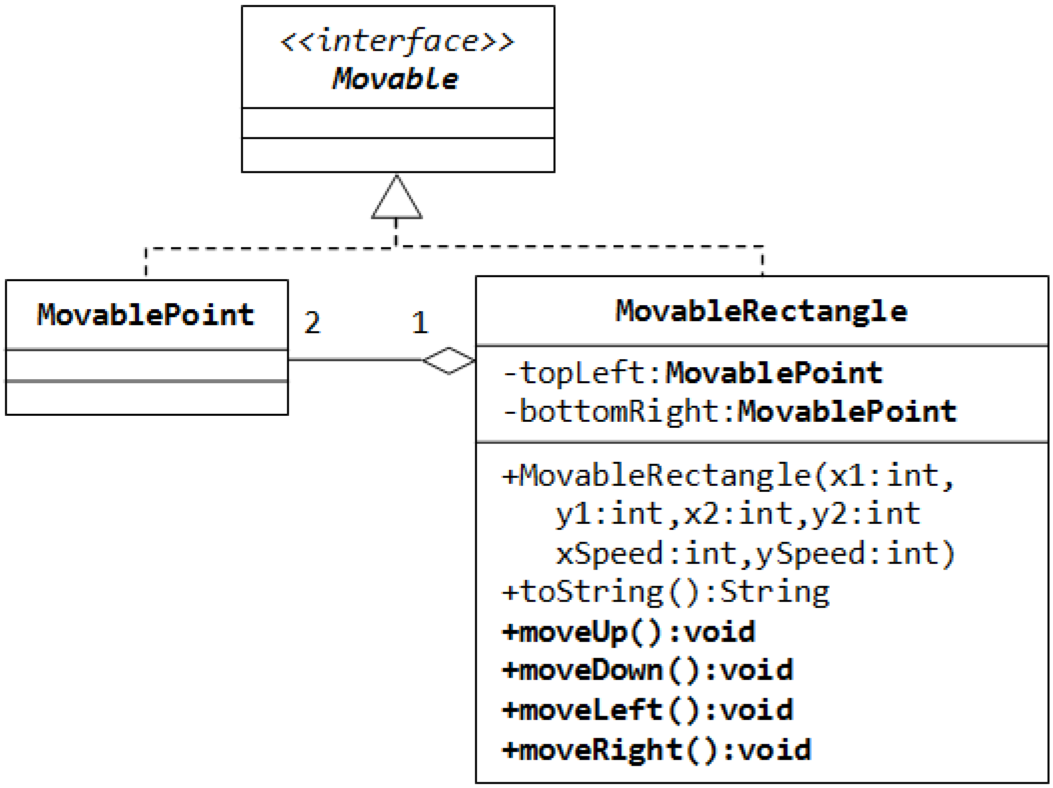
```
public class MovablePoint implements Movable { // saved as
"MovablePoint.java"
   // instance variables
   int x, y, xSpeed, ySpeed;     // package access

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

```
   }
   ......

   // 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);     // upcast
System.out.println(m1);
m1.moveLeft();
System.out.println(m1);

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

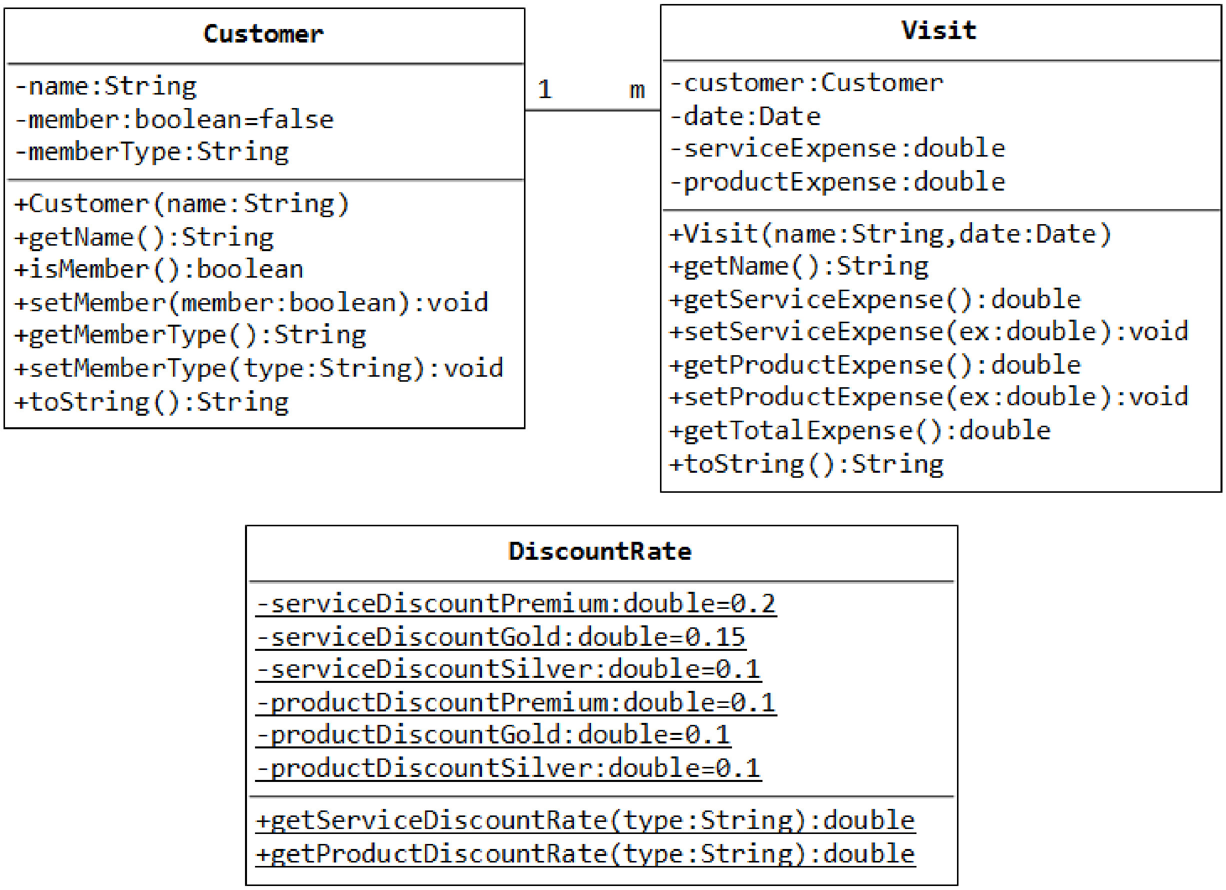



What is the difference between an interface and an abstract class?

## More Exercises on OOP

Exercise (Discount System): You are asked to write a discount system for a beauty saloon, which provides services and sells beauty products. It offers 3 types of memberships: Premium, Gold and Silver. Primium, gold and sivler members receive a discount of 20%, 15%, and 10%, respectively, for all services provided. Customers without membership receive no discount. All members receives a flat 10% discount on products purchased (this might change in future). Your system shall consist of three classes: `Customer`, `Discount` and `Visit`. It shall compute the total bill if a customer purchases $x of products and $y of services, for a visit. Also write a test program to exercise all the classes.

Hint:




DiscountRate contains `static` methods.