Lab 6. OOP - Inheritance

Writing Good Programs

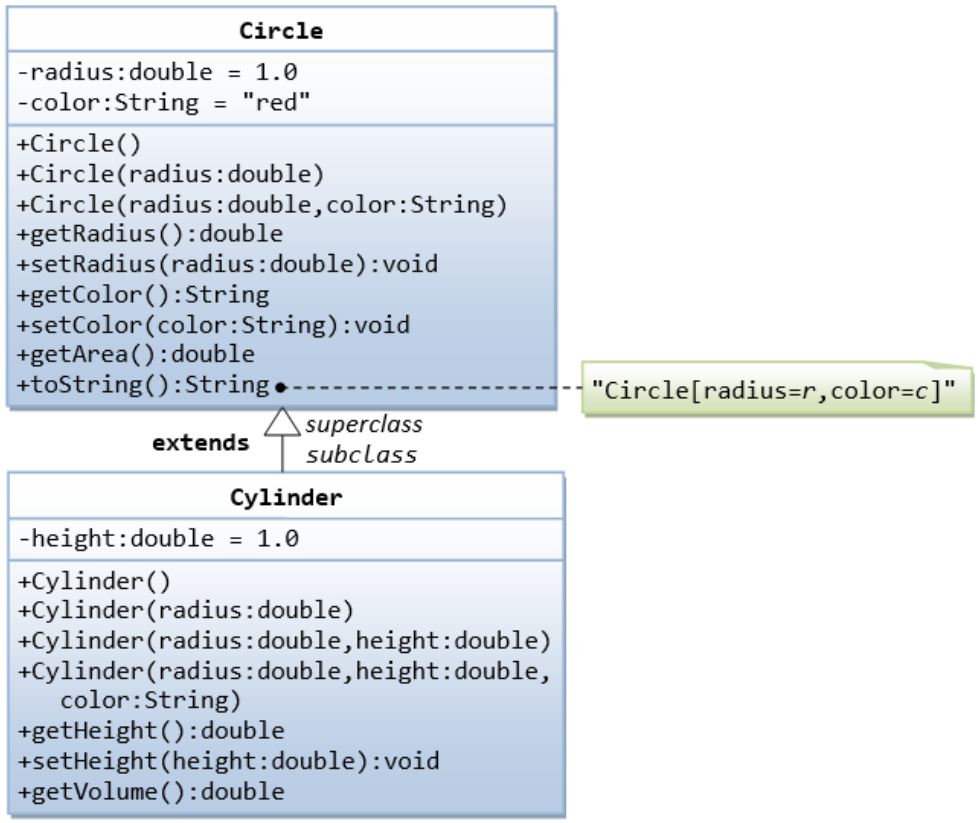
The only way to learn programming is program, program and program. Learning programming is like learning cycling, swimming or any other sports. You can't learn by watching or reading books. Start to program immediately. On the other hands, to improve your programming, you need to read many books and 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 Inheritance

1.1 An Introduction to OOP Inheritance by Example - The Circle and Cylinder Classes

This exercise shall guide you through the important concepts in inheritance.



In this exercise, a subclass called Cylinder is derived from the superclass Circle as shown in the class diagram (where an an arrow pointing up from the subclass to its superclass). Study how the subclass Cylinder invokes the superclass' constructors (via *super()* and *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.

```
1 public class Cylinder extends Circle {  // Save as "Cylinder.java"
    private double height;  // private variable
3
```

```
  // Constructor with default color, radius and height
  public Cylinder() {
    super();        // call superclass no-arg constructor Circle()
    height = 1.0;
  }
  // Constructor with default radius, color but given height
  public Cylinder(double height) {
    super();        // call superclass no-arg constructor Circle()
    this.height = height;
  }
  // Constructor with default color, but given radius, height
  public Cylinder(double radius, double height) {
    super(radius);  // call superclass constructor Circle(r)
    this.height = height;
  }

  // A public method for retrieving the height
  public double getHeight() {
    return height;
  }

  // A public method for computing the volume of cylinder
  //  use superclass method getArea() to get the base area
  public double getVolume() {
    return getArea()*height;
  }
}
```

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

```
public class TestCylinder {  // save as "TestCylinder.java"
  public static void main (String[] args) {
    // Declare and allocate a new instance of cylinder
    //   with default color, radius, and height
    Cylinder cylinder1 = new Cylinder();
    System.out.println("Cylinder:"
      + " radius=" + cylinder1.getRadius()
      + " height=" + cylinder1.getHeight()
      + " base area=" + cylinder1.getArea()
      + " volume=" + cylinder1.getVolume());

    // Declare and allocate a new instance of cylinder
    //   specifying height, with default color and radius
    Cylinder cylinder2 = new Cylinder(10.0);
    System.out.println("Cylinder:"
      + " radius=" + cylinder2.getRadius()
      + " height=" + cylinder2.getHeight()
```

```
        + " base area=" + cylinder2.getArea()
        + " volume=" + cylinder2.getVolume());

      // Declare and allocate a new instance of cylinder
      //   specifying radius and height, with default color
      Cylinder cylinder3 = new Cylinder(2.0, 10.0);
      System.out.println("Cylinder:"
        + " radius=" + cylinder3.getRadius()
        + " height=" + cylinder3.getHeight()
        + " base area=" + cylinder3.getArea()
        + " volume=" + cylinder3.getVolume());
   }
}
```

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 ($= 2\pi \times radius \times height + 2 \times base - area$) 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;
}
```

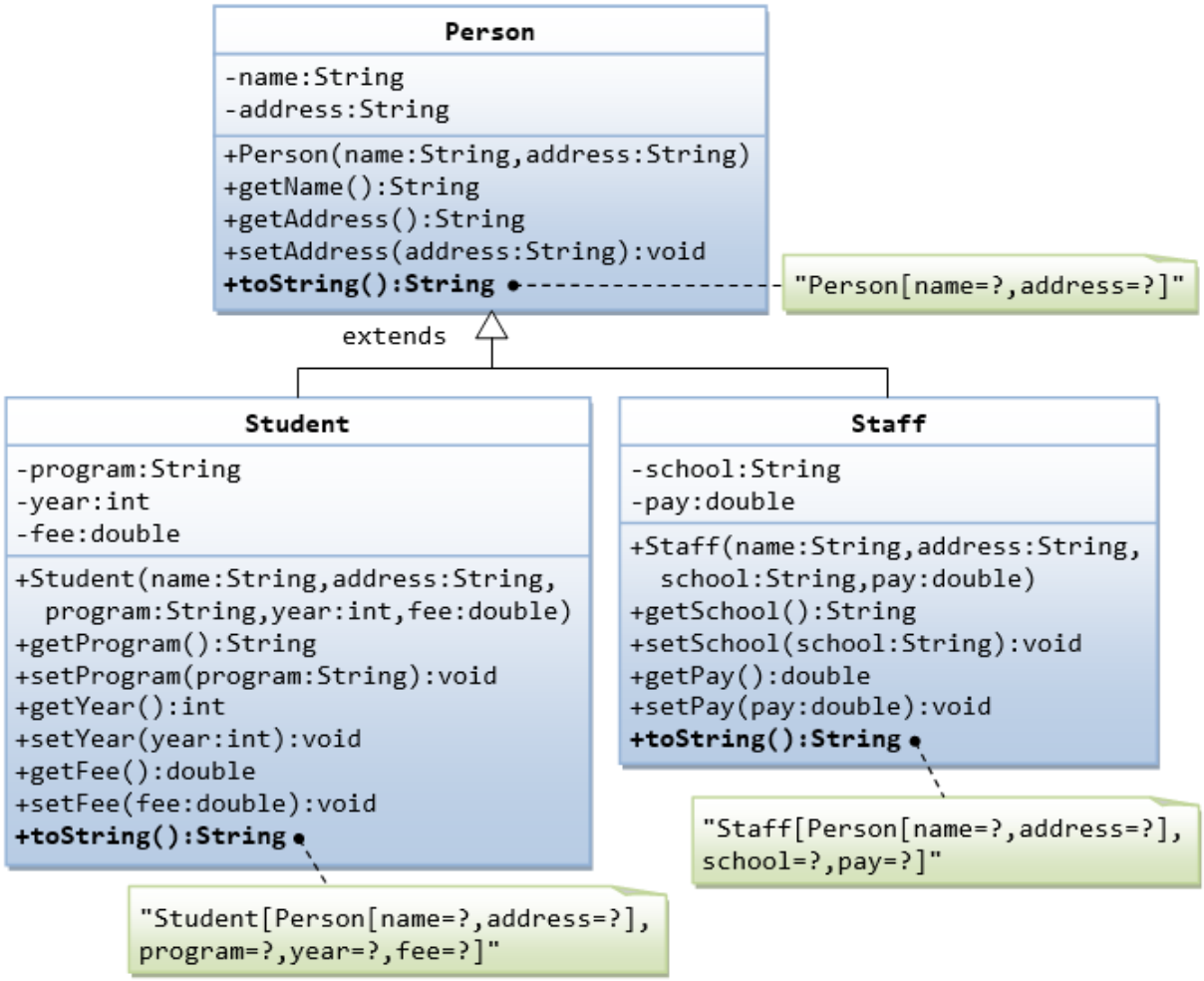
Try out the *toString()* method in TestCylinder.

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.

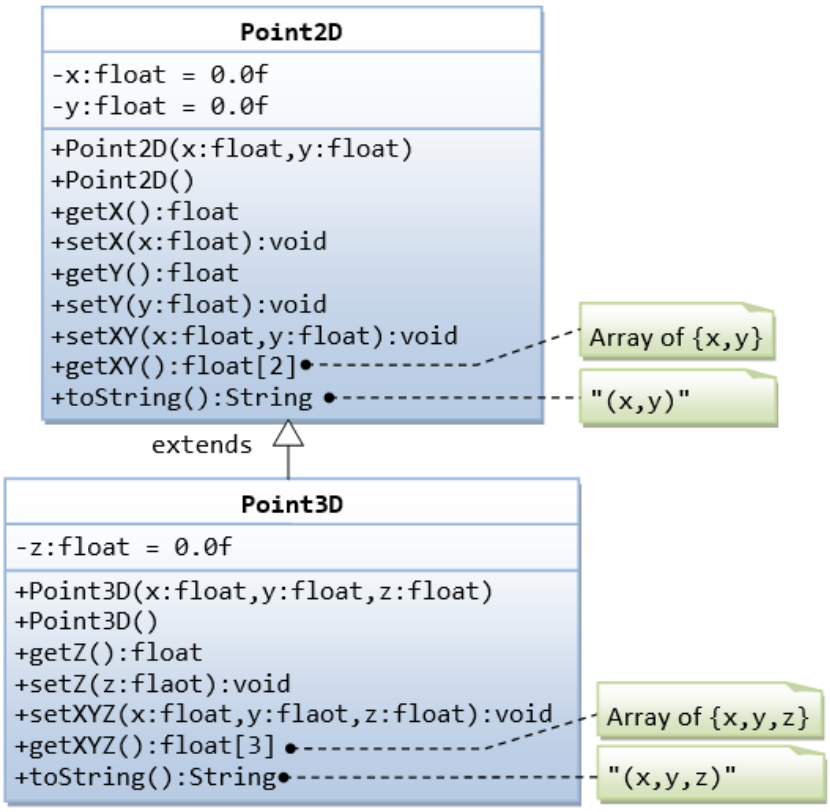
1.2 Superclass Person and its subclasses

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



1.3 Point2D and Point3D

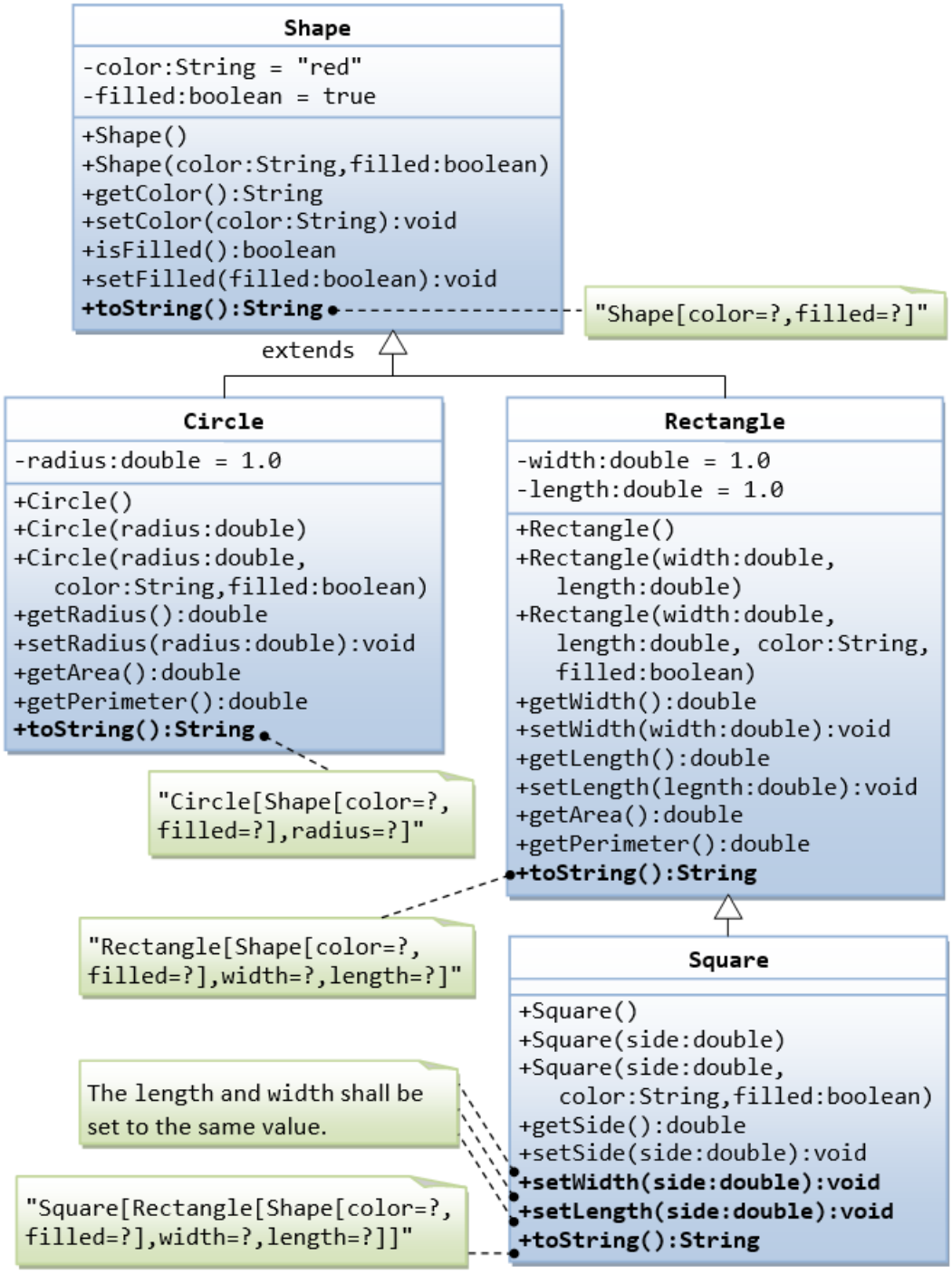
Write the classes as shown in the following class diagram. Mark all the overridden methods with annotation @Override.



Hints

1. You cannot assign floating-point literal say 1.1 (which is a *double*) to a *float* variable, you need to add a suffix f, e.g. $0.0f$, $1.1f$.
2. The instance variables x and y are private in Point and cannot be accessed directly in the subclass MovablePoint. You need to access via the public getters and setters. For example, you cannot write $x \mathrel{+}= xSpeed$, you need to write $setX(getX() + xSpeed)$.

1.4 Superclass Shape and its 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).

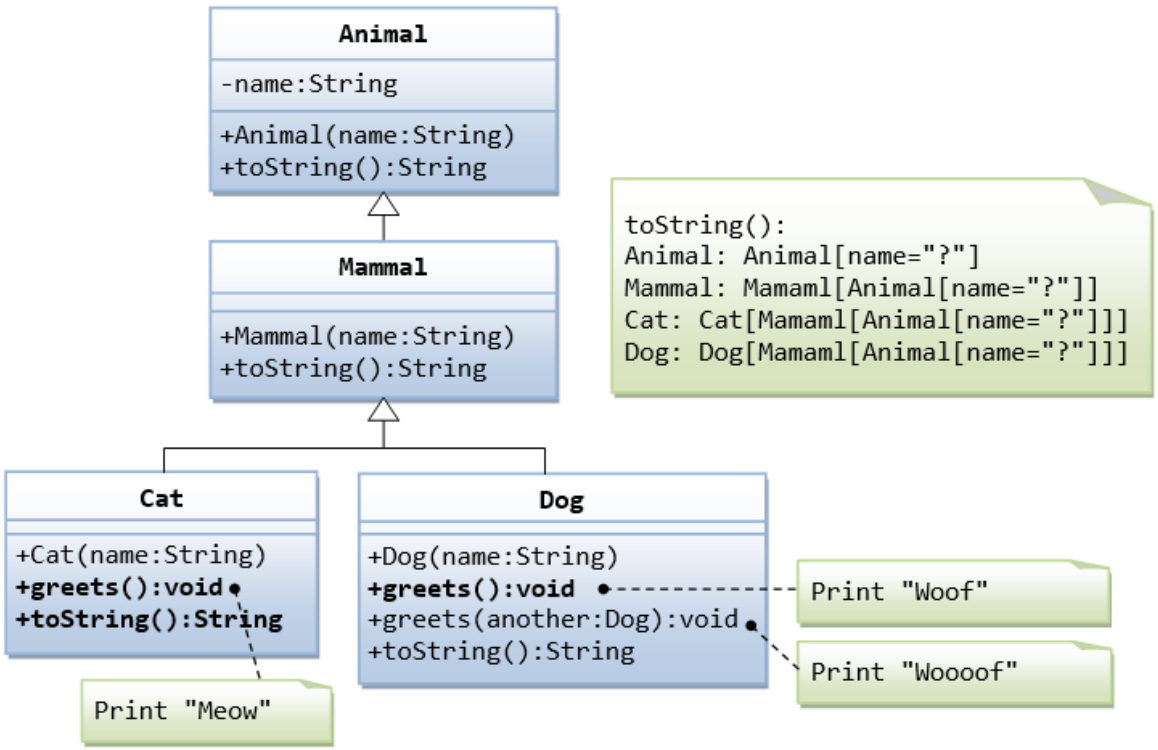
Hint:

```java
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.

1.5 Superclass Animal and its subclasses

Write the codes for all the classes as shown in the class diagram.



2 Exercises on Composition vs Inheritance

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

2.1 The Point and Line Classes

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).

```java
/**
 * Point.java
 */
public class Point {
  // Private variables
  private int x;     // x co-ordinate
  private int y;     // y co-ordinate

  // Constructor
  public Point (int x, int y) {......}

  // Public methods
  public String toString() {
    return "Point: (" + x + "," + y + ")";
  }

  public int getX() {......}
  public int getY() {......}
  public void setX(int x) {......}
  public void setY(int y) {......}
  public void setXY(int x, int y) {......}
}
```

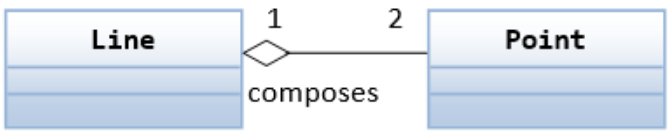
```java
/**
 * TestPoint.java
 */
public class TestPoint {
  public static void main(String[] args) {
    Point p1 = new Point(10, 20);   // Construct a Point
    System.out.println(p1);
    // Try setting p1 to (100, 10).
```

```
    ......
  }
}
```

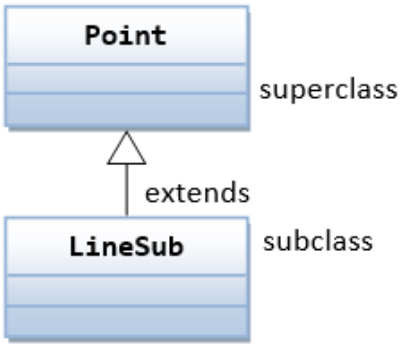
```
/**
 * Line.java
 */
public class Line {
  // A line composes of two points (as instance variables)
  private Point begin;    // beginning point
  private Point end;      // ending point

  // Constructors
  public Line (Point begin, Point end) {  // caller to construct the Points
    this.begin = begin;
    ......
  }
  public Line (int beginX, int beginY, int endX, int endY) {
    begin = new Point(beginX, beginY);   // construct the Points here
    ......
  }

  // Public methods
  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
  // Math.sqrt(xDiff*xDiff + yDiff*yDiff)
  public double getGradient() { ..... } // Gradient in radians
  // Math.atan2(yDiff, xDiff)
}
```

```java
/**
 * TestLine.java
 */
public class TestLine {
  public static void main(String[] args) {
    Line l1 = new Line(0, 0, 3, 4);
    System.out.println(l1);

    Point p1 = new Point(...);
    Point p2 = new Point(...);
    Line l2 = new Line(p1, p2);
    System.out.println(l2);
    ...
  }
}
```

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:

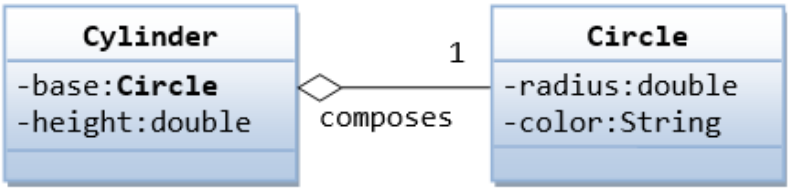


Let's re-design the Line class (called LineSub) as a subclass of class Point. LineSub inherits the starting point from its superclass Point, and adds an ending point. Complete the class definition. Write a testing class called TestLineSub to test LineSub.

```java
/**
 * LineSub.java
 */
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?

2.2 The Circle and Cylinder Classes Using Composition



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

```java
/**
 * Cylinder.java
 */
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?