Metodología de la programación

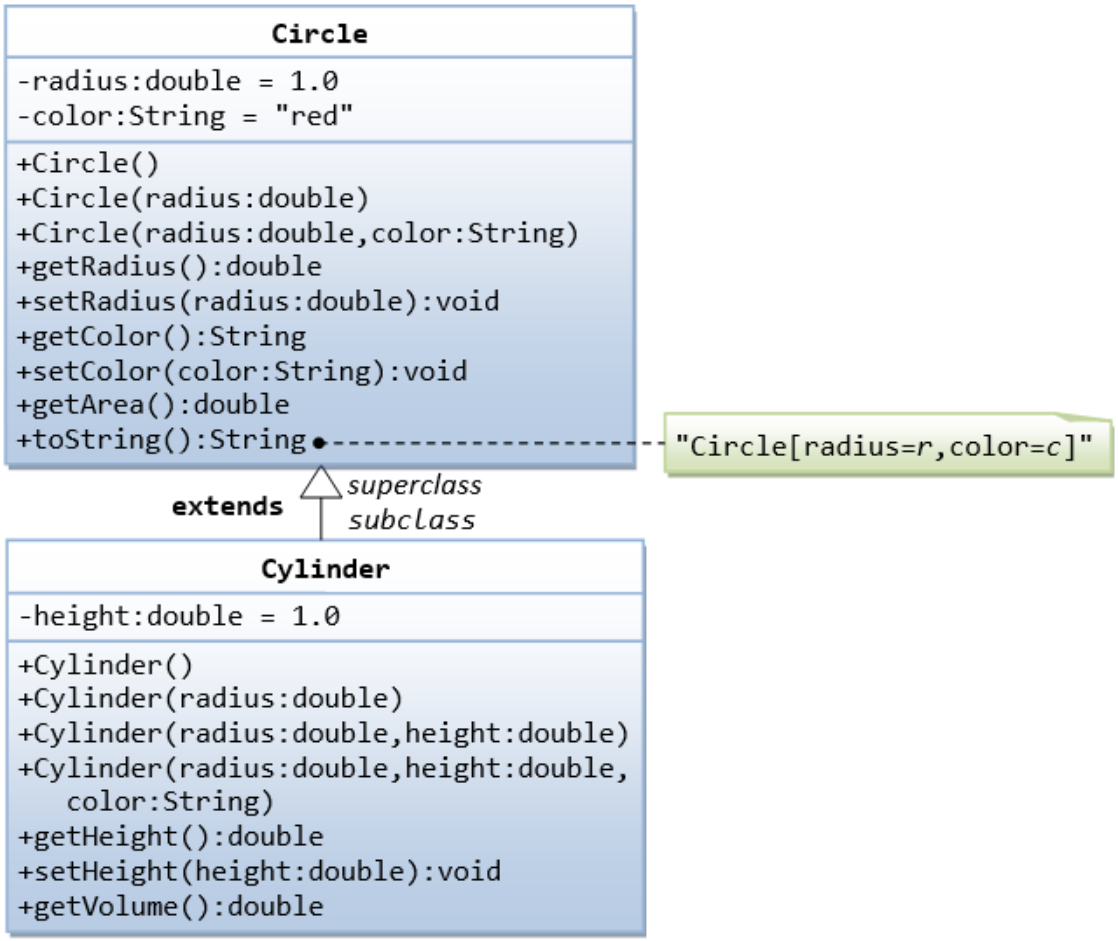
Ejercicios de Programación Orientada a Objetos

Estos ejercicios están basados en los ejercicios propuestos en la Universidad Tecnológica de Nanyang en Singapur.

3.Ejercicios sobre herencia de clases

3.1 Ejercicios introductorios – Las clases `Circle` y `Cylinder`

En este ejercicio exploraremos conceptos básicos sobre herencia entre clases de objetos.



En el ejercicio crearemos una ***subclase*** llamada `Cylinder` que se deriva de la ***superclase*** `Circle` tal como se muestra en el diagrama. Las subclases heredan de las superclases sus atributos y métodos, y se remarcan con un triángulo, que puede dibujarse como punta de flecha grande y abierta o en medio de las lineas de conexión.

Fíjate cómo la subclase `Cylinder` invoca al constructor de la superclase utilizando el método super: t `super()` y `super(radius)`) y hereda los atributos y métodoss de la super clase `Circle`.

Para poder hacer el ejercicio, puedes reutilizar la clase `Circle` que creaste, ya que es la misma que en este diagrama. La idea de la herencia de clases es precisamente esa: no volver a escribir código ya escrito. Para ello, el diseño de las clases debe hacerse con cabeza. No repetir código tiene varias ventajas: no trabajas doble, no duplicas los errores, las correcciones sólo se hacen en un sitio.

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

```
    // 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;
    }
}
```

Prueba la ejecución de este programa paso a paso, y verifica cómo llama a los distintos métodos entre las dos clases anteriores realizando una ejecución paso a paso en depuración:

```
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 c1 = new Cylinder();
        System.out.println("Cylinder:"
                + " radius=" + c1.getRadius()
                + " height=" + c1.getHeight()
                + " base area=" + c1.getArea()
                + " volume=" + c1.getVolume());

        // Declare and allocate a new instance of cylinder
        //   specifying height, with default color and radius
```

```
        Cylinder c2 = new Cylinder(10.0);
        System.out.println("Cylinder:"
                + " radius=" + c2.getRadius()
                + " height=" + c2.getHeight()
                + " base area=" + c2.getArea()
                + " volume=" + c2.getVolume());

        // Declare and allocate a new instance of cylinder
        //   specifying radius and height, with default color
        Cylinder c3 = new Cylinder(2.0, 10.0);
        System.out.println("Cylinder:"
                + " radius=" + c3.getRadius()
                + " height=" + c3.getHeight()
                + " base area=" + c3.getArea()
                + " volume=" + c3.getVolume());
    }
}
```

Sobrecarga de métodos y "super": La sobrecarga de métodos permite que un mismo método sea definido en dos clases relacionadas por herencia. La superclase tendrá una definición e implementación del método, mientras que la subclase tendrá otra. El problema en este caso es cómo hacer que la subclase pueda invocar al método de la superclase (no siempre es necesario, pero suele ser interesante poder hacerlo). Para ello tenemos la palabra reservada "super", que se refiere a la parte de la instancia definida por la superclase.

En el ejercicio, la subclase `Cylinder` hereda el método `getArea()` de su superclase.
Intenta sobercargar el método `getArea()` de la subclase `Cylinder` para calcular el área de la superficie del cilindro (=2π×radio×altura + 2×areadelabase) en vez del área de la base. De esta forma, si un programa llama al método `getArea()` de un círculo, devolverá el área del círculo. Si se llama a `getArea()` de un cilindro, devolverá el resultado adecuado.

Cuando haces esa modificación… ¿qué sucede con el método `getVolume()`? ¿Por qué?

Si se sobrecarga `getArea()` de la subclase `Cylinder`, entonces `getVolume()` deja de funcionar. Esto se debe a que `getVolume()` va a usar la nueva `getArea()` sobrecargada, ya que está en su misma clase. Si no lo hubiésemos sobrecargado, Java iría a buscar el método en su superclase, y así sucesivamente, hasta que lo encontrase.

¿Cómo lo arreglamos?.
Modificando `getVolume()` para que use el `getArea()` de la superclase.
Pista: Si quieres usar el método `getArea()` de la clase actual no hay que hacer nada, pero si quieres usar el de la superclase, hay que utilizar super: `super.getArea()` usado en el método `getVolume()` llamará al método `getArea()` de la superclase.

Ahora tú: Intenta modificar el método `toString()` de la clase `Cylinder` para que use el método de la superclase.

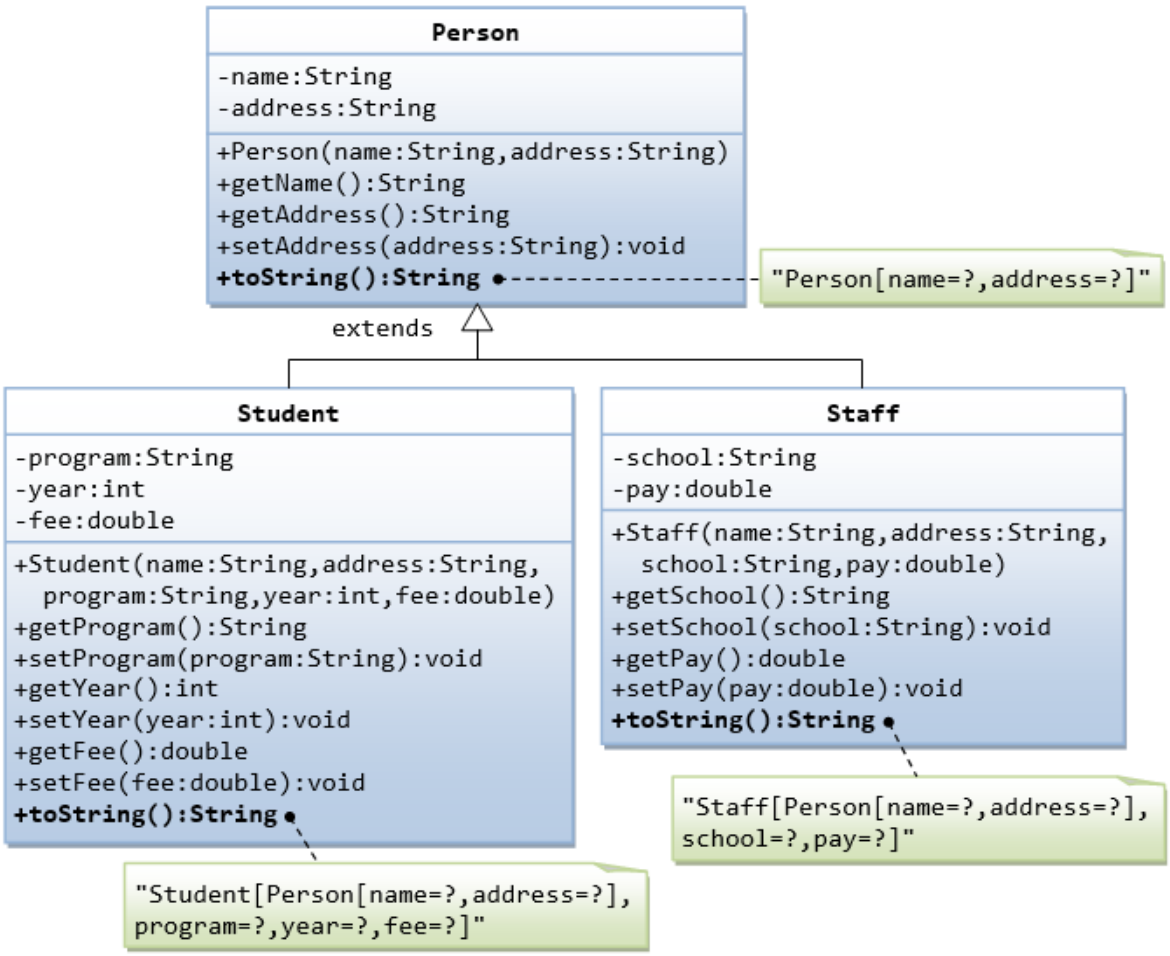
```
@Override
public String toString() {      // in Cylinder class
    return "Cylinder: subclass of " + super.toString()  // use Circle's toString()
            + " height=" + height;
}
```

¿Has visto eso de `@Override` ?
En java, los elementos que llevan un @ delante y están fuera de los métodos y clases se les denomina "anotaciones". Una anotación sirve para guiar al compilador a la hora de realizar su tarea y para añadir código a nuestro código para refinar el funcionamiento de nuestro programa.
En el caso de `@Override` lo que se está indicando es que el compilador Java debe asegurarse de que existe un método con la misma firma que el que estamos definiendo en la superclase de la clase que estamos definiendo. Esto ayuda a no equivocarnos al escribir los nombres y firmas de los métodos a redefinir: Imagina que no tuviésemos esa anotación y escribiésemos `public String ToString()`... el programa compilaría y funcionaría, pero cuando quisiésemos sacar algo por consola, no saldría lo que nosotros esperábamos, y nos volveríamos locos buscando el porqué: he escrito `ToString` con la T mayúscula, cuando debería ser minúscula.
Al usar `@Override`, lo que estamos haciendo es asegurarnos que hemos redefinido bien el método: Java nos avisaría de que hay un error, porque el método `ToString` con T mayúscula no existe en la superclase.

Como puedes imaginar, usar @Override no es obligatorio, pero sí que es una buena ayuda, y su uso se considera una **buena práctica**.

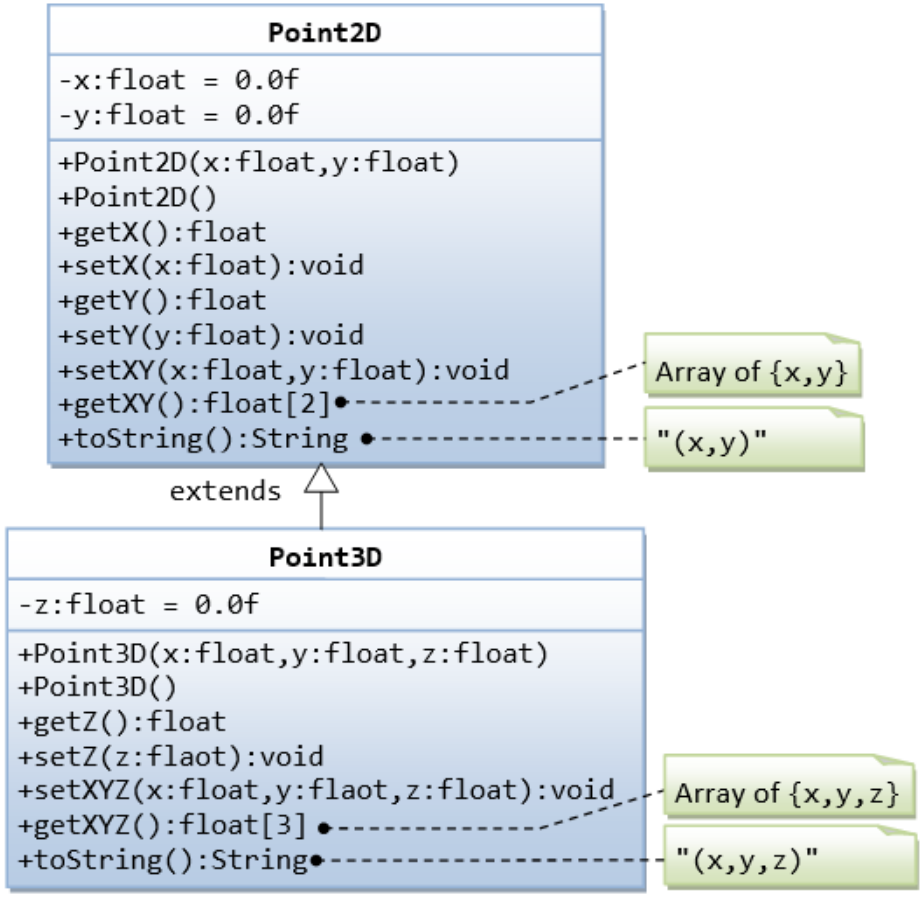
3.2 Ejercicio: superclase `Person` y sus subclases

Escribe las clases del siguiente diagrama, marcando como @Override todos los métodos que se sobrecargen.



3.3 Ejercicio: `Point2D` y `Point3D`

Escribe las clases del siguiente diagrama, marcando como @Override todos los métodos que se sobrecargen.



Notas:

1. No se puede asignar literales de punto flotante de manera directa (si escribes 1.1, estás escribiendo un double, no un float). Para poder asignar un float debes añadir una f al final: 1.1f.
2. Como se han marcado los atributos x e y como privados, no se puede acceder a ellos desde la subclase: hay que usar los getters y setters correspondientes. Por ejemplo:

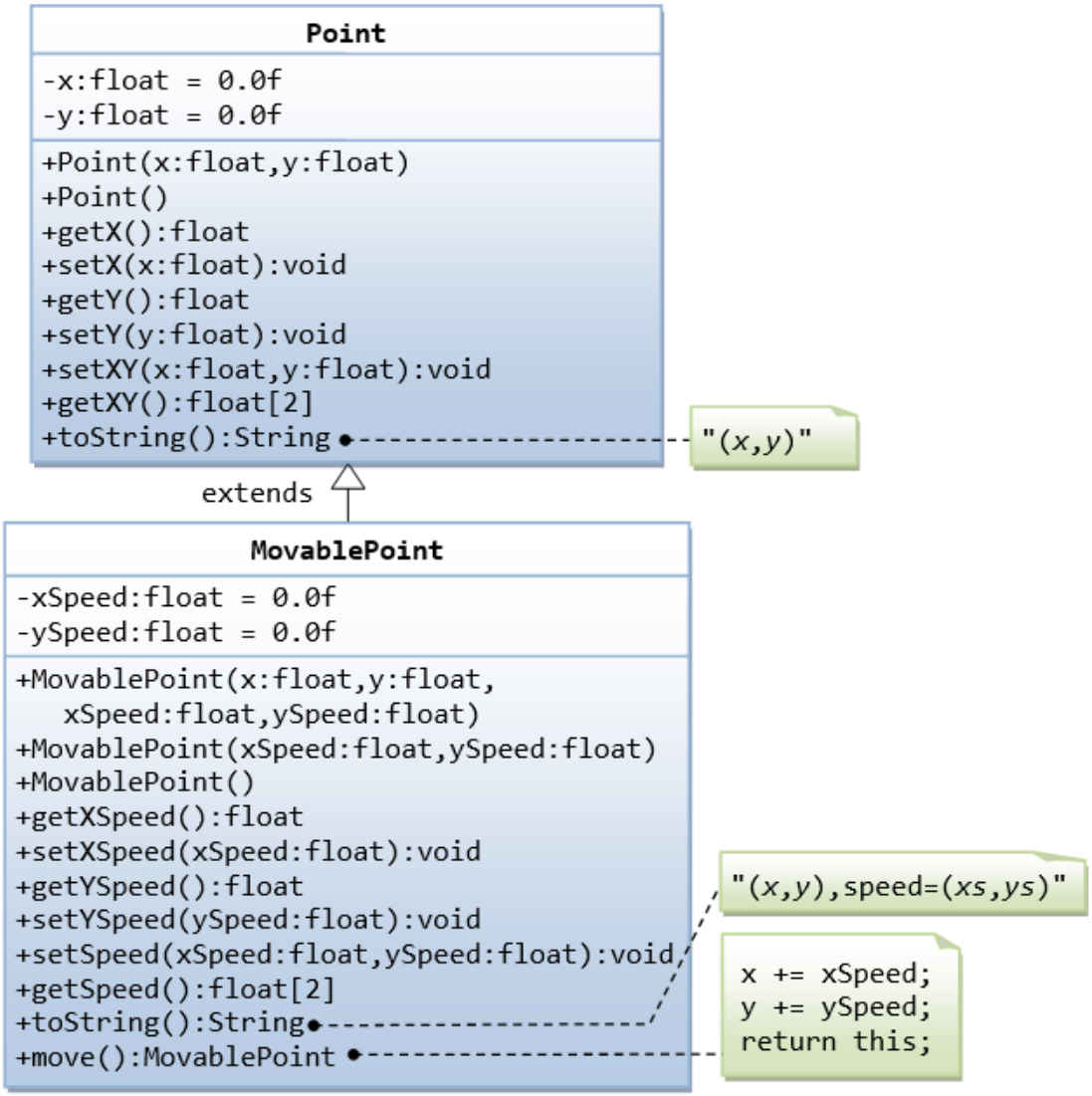
```
public void setXYZ(float x, float y, float z) {
    setX(x);     // or super.setX(x), use setter in superclass
    setY(y);
    this.z = z;
}
```

3. El método `getXY()` debe devolver un array de `float`:

```
public float[] getXY() {
    float[] result = new float[2];  // construct an array of 2 elements
    result[0] = ...
    result[1] = ...
    return result;  // return the array
}
```

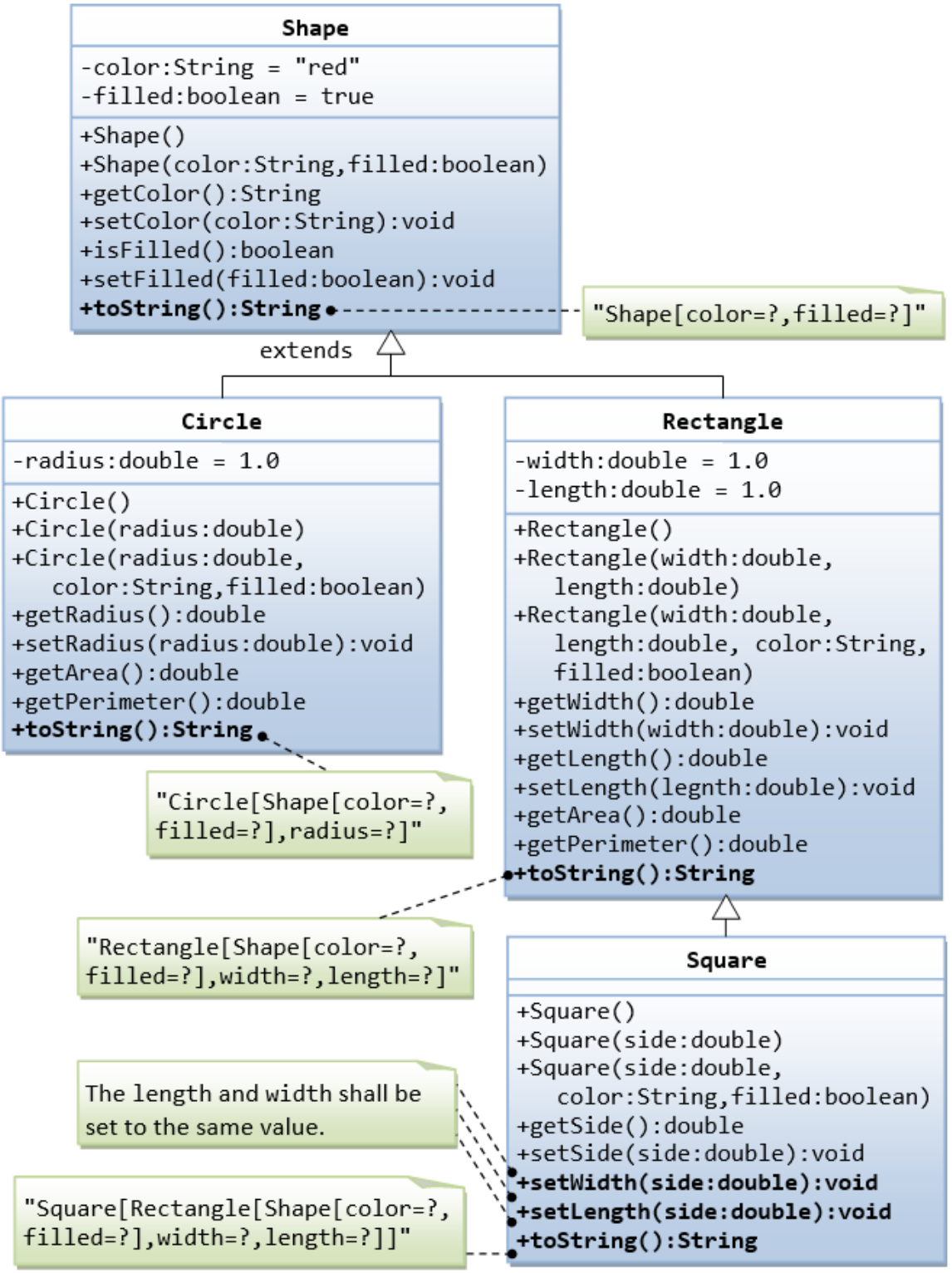
3.4 Ejercicio: `Point` y `MovablePoint`

Escribe las clases del siguiente diagrama, marcando como @Override todos los métodos que se sobrecargen.



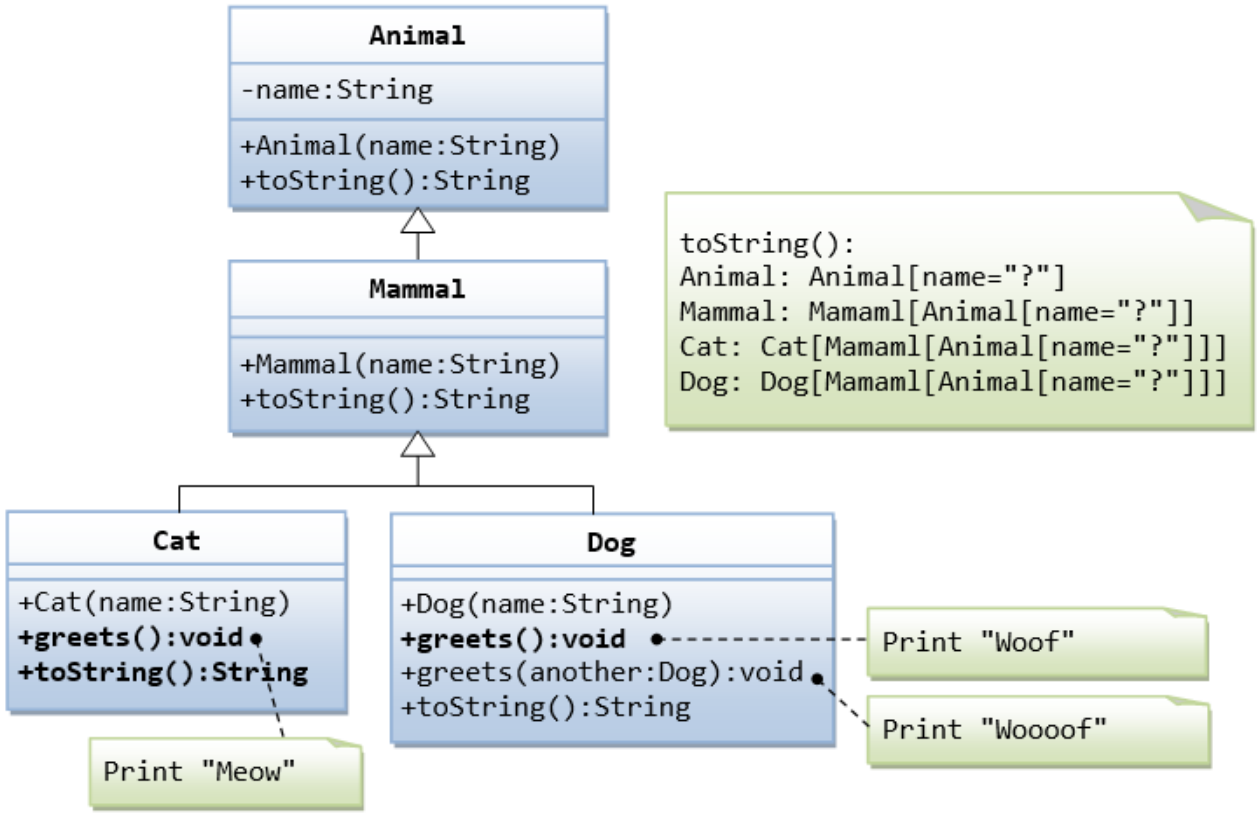
3.5 Ejercicio: superclase `Shape` y sus subclases `Circle`, `Rectangle` y `Square`

Escribe las clases del siguiente diagrama, marcando como @Override todos los métodos que se sobrecargen.



3.6 Ex: Superclass `Animal` and its subclasses

Escribe las clases del siguiente diagrama, marcando como @Override todos los métodos que se sobrecargen.



Construye un programa que pruebe las clases anteriores.

¿Puedo hacer una llamada a greets() de un animal? ¿Por qué?

4.Diferencias entre la composición y la herencia

La composición y la herencia son dos estrategias para la reutilización de código, pero cada una presenta diferencias notables.

4.1 Las clases `Point` y `Line`

Empecemos con la composición: "Una línea se compone de dos puntos".

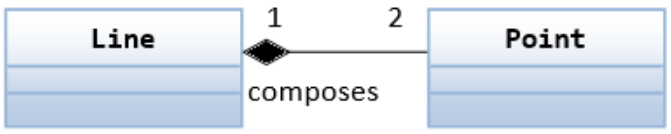
Partiendo de esta frase definitoria, las clases `Point` y `Line` quedarían de la siguiente forma:

```
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) {......}
}
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).
        ......
    }
}
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)
}
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);
        ...
    }
```

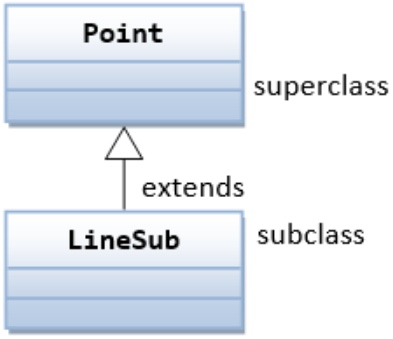
```
}
```

El diagrama de clases quedaría así:



En la composición hacemos uso de otros objetos como atributos de una de las clases. Esto permite que se varíe la estructura en tiempo de ejecución (en algunos casos) mediante el uso de listas de objetos que pueden crecer y decrecer. En el caso que nos ocupa, eso no sucede, ya que se especifica que una línea se compone de dos puntos: sin esos puntos, no se puede componer. Cosa distinta es la agregación, que sí permite la variación de esos elementos. (NOTA: composición = rombo relleno, agregación = rombo en blanco).

Si ahora intentamos el diseño mediante la herencia, una clase `Line` heredará, de manera que la definición del problema quedará como : "Una linea es un punto extendido por otro punto", o dicho de otro modo: una línea tiene un punto de origen, que hereda del punto, e internamente tiene un punto de fin:



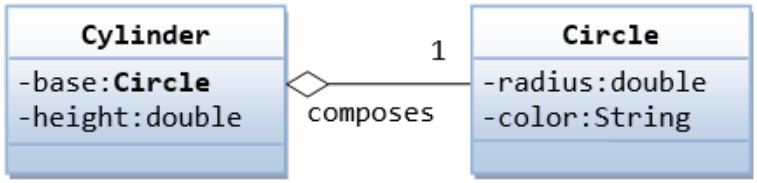
Dado que no queremos escribir código duplicado, seguiremos usando la clase `Point`, esta vez como superclase, y definiremos una nueva subclase denominada `LineSub` a la que añadadiremos un punto final. Completa el trabajo añadiendo los métodos necesarios y el programa de prueba.

```
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
}
```

Como puedes ver, existen distintas maneras de diseñar un mismo problema, y dependiendo del mismo, unas pueden ser mejores que otras según distintos parámetros: Para el caso que nos ocupa, ¿cuál consideras mejor? ¿por qué?

4.2 Las clases `Circle` y `Cylinder` usando composición:



Intenta redefinir las clases anteriores de ejercicios previos mediante la composición usando la frase definitoria: "Un cilindro está compuesto de un círculo base y una altura".

```
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;
    }
    ......
}
```

¿Qué diseño es mejor en este caso? ¿Por qué?

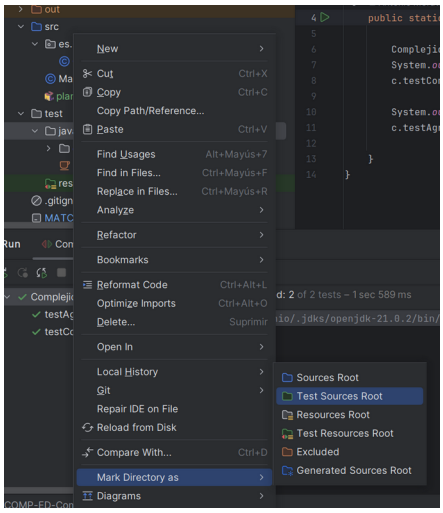
5.1 Mejorando la calidad de las clases

Dado que tenemos los programas de prueba de las clases anteriores (bloques 3 y 4, todos los contenidos en este documento) , y en estos programas se muestra qué se espera que aparezca en pantalla según la ejecución de los ejercicios, añada para todas las clases de los ejercicios los tests necesarios para verificar la correcta ejecución de todos ellos.

Asegure que los tests tienen una cobertura del 100% del código desarrollado en las clases.

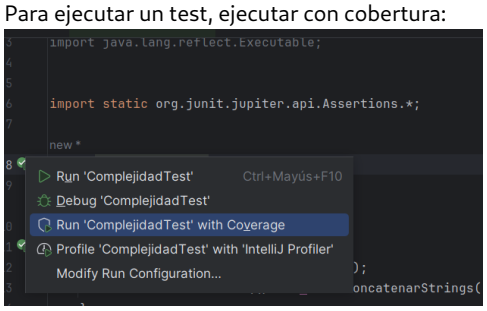
(Usamos JUnit 5)
Pasos:
Crear directorio test
Crear directorio test/java
Crear directorio test/resources

Marcar directorio test/java como “Test Source Root”
Marcar directorio test/resources como “Test Resources Root”



Sobre la clase a testear, botón derecho → “Generate” y en la ventana que aparece, elegir “Test”.

Para ejecutar un test, ejecutar con cobertura:



Referencia:
https://www.jetbrains.com/help/idea/tests-in-ide.html
https://www.jetbrains.com/help/idea/testing.html