# SOFTWARE 2 PRACTICAL

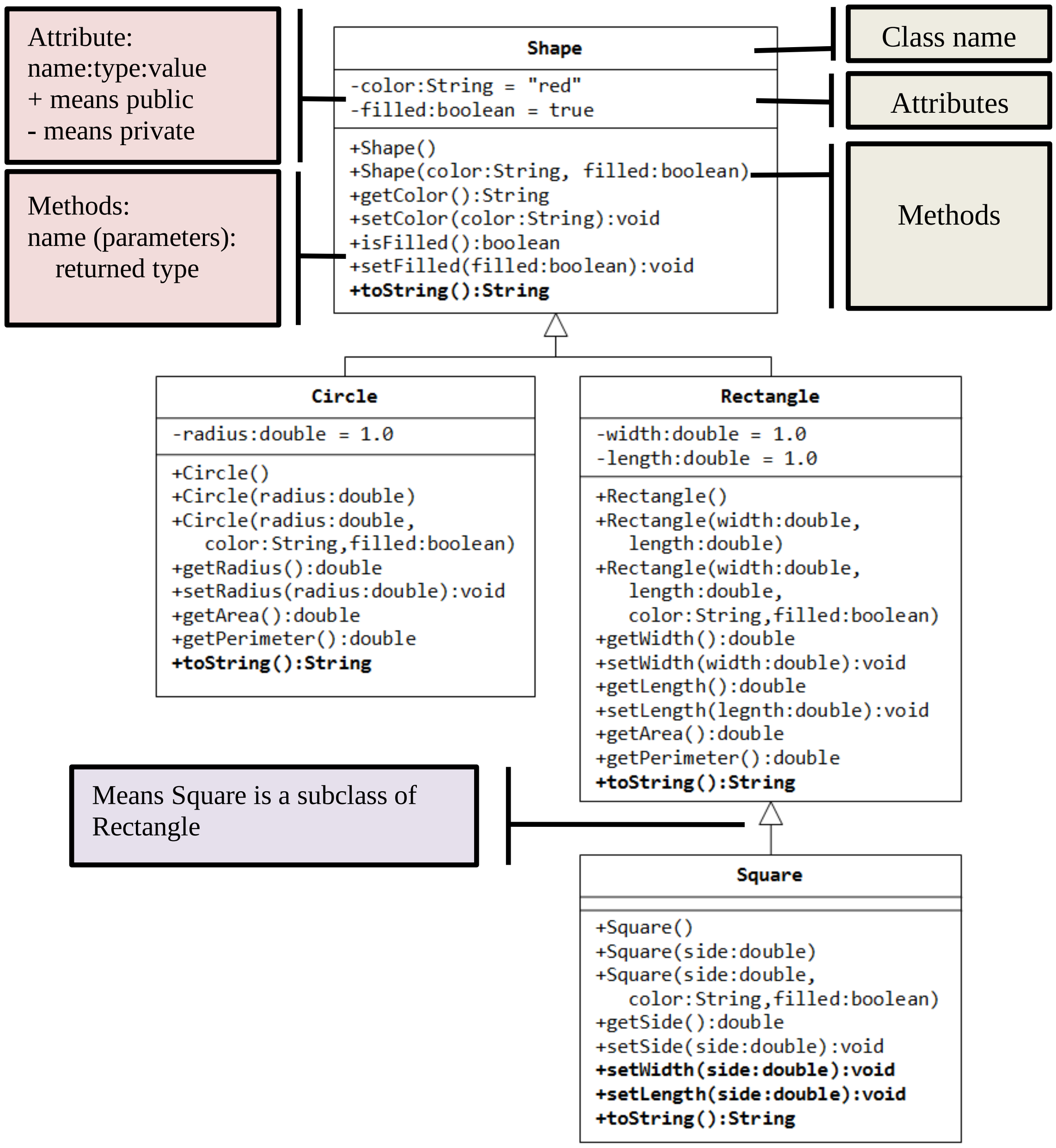
## SIMPLE INHERITANCE

Week 5 – Practical 4

The aim of this week practical is to understand the basic concept of inheritance in the OOP paradigm. Next week we will be looking at more advanced concept of inheritance based on today's practical. So it is important that you finish this practical before next week practical.

## Problem:

We want to build a program that draws shapes on a canvas, and in addition provides additional information about the shapes being drawn. We will start with few simple shapes like circles, rectangles and squares. We provide the following UML diagram for the class hierarchy.



Write a superclass called `Shape` (as shown in the class diagram), which contains:

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

ii. Two constructors: a no-arg (no-argument) constructor that initializes the `color` to "red" and `filled` to `true`, and a constructor that initializes the `color` and `filled` to the given values.

iii. 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).

iv. A `toString()` method that returns "`A Shape with color of xxx and filled/Not filled`".

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

The `Circle` class contains:

i. An instance variable `radius` (type is `double`).

ii. Three constructors as shown. The no-arg constructor initializes the radius to `1.0`.

iii. Getter and setter for the instance variable `radius`.

iv. Methods `getArea()` and `getPerimeter()`.

v. 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:

vi. Two instance variables `width` and `length` (both `double`).

vii. Three constructors as shown. The no-arg constructor initializes the `width` and `length` to `1.0`.

viii. Getter and setter for all the instance variables.

ix. Methods `getArea()` and `getPerimeter()`.

x. 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 modelled as a subclass of `Rectangle`. `Square` has no instance variable, but inherits the instance variables width and length from its superclass Rectangle.

xi. Provide the appropriate constructors (as shown in the class diagram).

xii. 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.

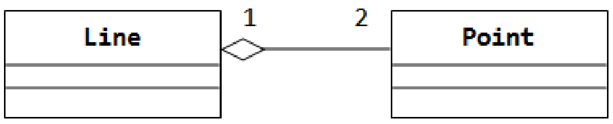
xiii. Do you need to override the `getArea()` and `getPerimeter()`? Try them out.

xiv. Override the `setLength()` and `setWidth()` to change both the `width` and `length`, so as to maintain the square geometry.

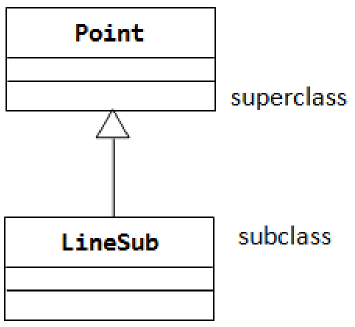
## Extension 1:

We want to add two additional classes Point and Line, and we need to make a design decision. Should we use ***composition*** or ***inheritance***?

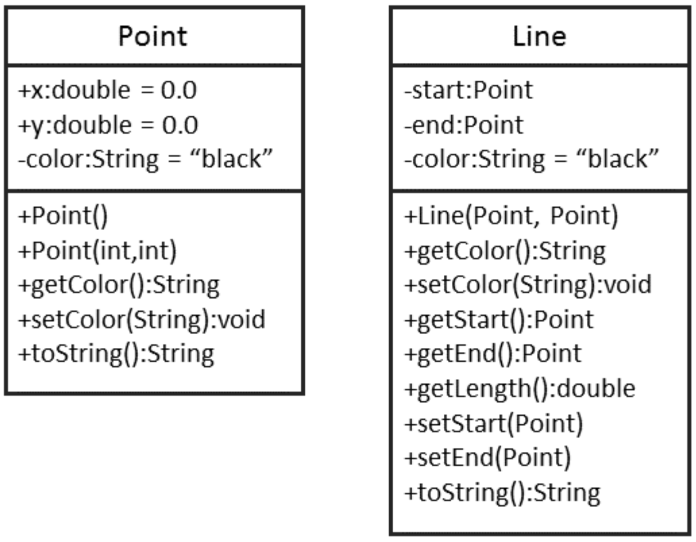
- Composition (has-a relationship):



- Inheritance (is-a relationship):



Choose which design is better suited for our problem and implement the two classes given the UML below:



## Extension 2:

For the moment, we have a description of a shape, but we have not thought about its location and orientation in space.

- How would you solve that issue?
- Should the attributes be public, private or protected?
- What kind of additional method do you need? Try to think about it as if you were implementing a graphical tool to draw such shapes.
- Finally, refactor the toString() method to get a better representation of the object.

Modify your classes accordingly.

## Extension 3: *Challenging*

**We do not need this for next week practical.**

We want to add more complex shapes to our library. Implement a class Triangle, a class Diamond, and a more interesting and challenging class Polygon for simple polygons (that is polygons where edges do not cross). The class Polygon is composed of a list of points stored in an ArrayList<Point>. All the classes should have the same methods (if possible) to the classes defined earlier.

Before implementing anything, where would you put the class Polygon in your class hierarchy? Is it a subclass of any existing classes, is there any existing classes that should be a subclass of Polygon? This is a question about design, and you need time to reflect on possible issues before you implement anything.

## Extension 4: *Very dfficult*

**We do not need this for next week practical.**

Finally, we would like all the shapes to have two more methods:

- public boolean contains(Point) which returns true if the point is inside the shape, false otherwise. Note, if the attribute `filled` is false, then the point must on the edges to return true, otherwise it should return false. You can find a description of the Ray Casting algorithm at: https://en.wikipedia.org/wiki/Point_in_polygon.
- Public double distance(Point) which returns the distance between the point and the shape. If the attribute `filled` is true, then if the point is inside the shape or on the edges, the distance should be zero. If the attribute `filled` is false, then the distance should be the minimum distance between the point and any edges from that shape. You can find the formulae to compute the distance between a point and a line at https://en.wikipedia.org/wiki/Distance_from_a_point_to_a_line.