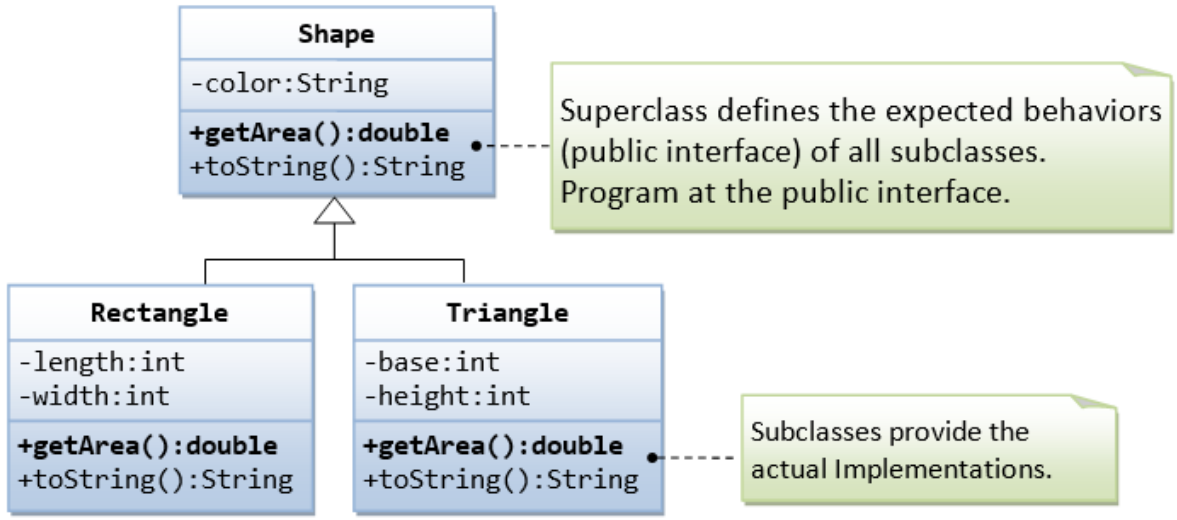
Exercise 09

Task 1 Polymorphism with Shapes:

Polymorphism is very powerful in OOP to *separate the interface (how it should look) and implementation* so as to allow the programmer to *program at the interface* in the design of a *complex system*.

Consider the following example. Suppose that our program uses many kinds of shapes, such as triangle, rectangle and so on. We should design a superclass called Shape, which defines the public interfaces (or behaviors) of all the shapes. For example, we would like all the shapes to have a method called `getArea()`, which returns the area of that particular shape. Below you find a class diagram of the shape class. Fill in the missing parts in the code too.



The parentclass `Shape.java`

```
/*
 * Superclass Shape maintain the common properties of all shapes
 */
public class Shape {
   // Add private member variable


   // Add Constructor

   @Override
   public String toString() {

   }

   // All shapes must have a method called getArea().
   public double getArea() {
      // We have a problem here!
      // We need to return some value to compile the program.
      System.err.println("Shape unknown! Cannot compute area!");
      return 0;
   }
}
```

Take note that we have a problem writing the `getArea()` method in the Shape class, because the area cannot be computed unless the actual shape is known. We shall print an error message for the time being. We will see later how to tackle this problem.
We can then implement the subclasses, such as `Triangle` and `Rectangle`, from the superclass Shape. Again fill in the missing code.

The childclass `Rectangle.java`

```
/*
 * The Rectangle class, subclass of Shape
 */
public class Rectangle extends Shape {
   // Add private member variables

   // Add Constructor

   @Override
   public String toString() {
   }

   // Override the inherited getArea() to provide the proper implementation area
//=length * width
   @Override
   public double getArea() {
   }
}
```

The childclass `Triangle.java`

```
/*
 * The Triangle class, subclass of Shape
 */
public class Triangle extends Shape {
   // Add Private member variables


   // Add Constructor, use chaining and super!

   @Override
   public String toString() {
   }

   // Override the inherited getArea() to provide the proper implementation,
//0.5*base*height
   @Override
   public double getArea() {
   }
}
```

The subclasses overrides the `getArea()` method inherited from the superclass, and provide the proper implementations for `getArea()`.

Test class (`TestShape.java`)

In our application, we create references of Shape, and assigne them instances of subclasses, as follows:

```
/*
 * A test class for Shape and its subclasses
```

```
 */
public class TestShape {
   public static void main(String[] args) {
      Shape s1 = new Rectangle("red", 4, 5);  // Upcast
      System.out.println(s1);  // Run Rectangle's toString()
      System.out.println("Area is " + s1.getArea());  // Run Rectangle's getArea()

      Shape s2 = new Triangle("blue", 4, 5);  // Upcast
      System.out.println(s2);  // Run Triangle's toString()
      System.out.println("Area is " + s2.getArea());  // Run Triangle's getArea()
   }
}
```

The expected outputs are:

```
Rectangle[length=4,width=5,Shape[color=red]]

Area is 20.0

Triangle[base=4,height=5,Shape[color=blue]]

Area is 10.0
```

The beauty of this code is that *all the references are from the superclass* (i.e., *programming at the interface level*). You could instantiate different subclass instances, and the code still works. You could extend your program easily by adding in more subclasses, such as `Circle`, `Square`, etc, with ease without changing too much of the super class.

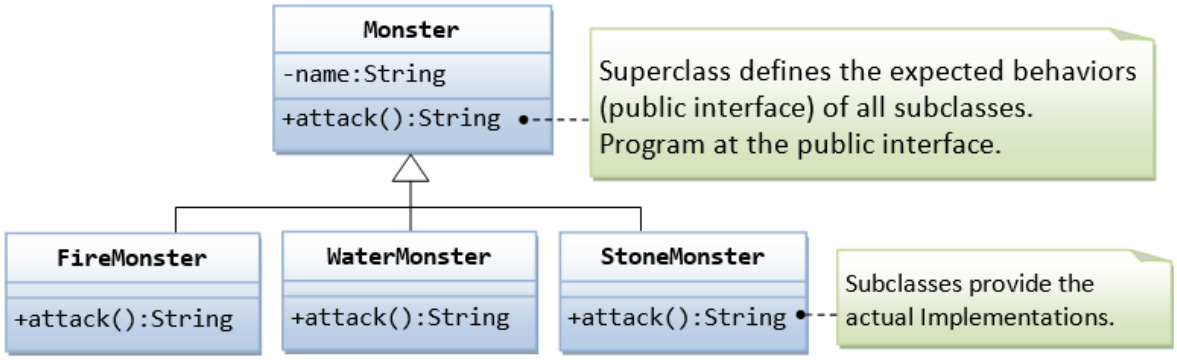
Nevertheless, the above definition of Shape class poses a problem, if someone instantiate a Shape object and invoke the `getArea()` from the Shape object, the program breaks. Try the following test code:

```
public class TestShape {
   public static void main(String[] args) {
      // Constructing a Shape instance poses problem!
      Shape s3 = new Shape("green");
      System.out.println(s3);
      System.out.println("Area is " + s3.getArea());  // Invalid output
   }
}
```

This is because the Shape class is meant to provide a common interface to all its subclasses, which are supposed to provide the actual implementation. We do not want anyone to instantiate a Shape instance. This problem can be resolved by using the so-called `abstract` class which we will learn in the next lecture!

Task 2 – Monster Polymorphism

In this example we create a game app, where we have many types of monsters that can attack. We will design a superclass called `Monster` and define the method `attack()` in the superclass. The subclasses then provides their actual implementation. In the main program, we declare instances of superclass, substituted with actual subclass; and invoke methods defined in the superclass. Below you find the class diagram for the monster class and the sub monsters.



Start with the implementation of the Monster class.
We need an instance variable name.
Then we also need a constructor.
Finally, we need an attack method. For the superclass return "Help!! I don't know how to attack!";
Next create three sub classes of monsters. FireMonster, WaterMonster and StroneMonster.
Override for each of them the attack method. (Fire “Attack with fire”, Water “Attack with water”, etc.)
Test your monsters in the following test class (TestMonster). **What can you learn from it?**
Extend the parent monster class with new methods or data fields that you think make sense (e.g., eat, attack power, etc.) be creative.
Write a test class and explore the behavior of your object monsters in the wild!

```
public class TestMonster {
   public static void main(String[] args) {
      // Program at the "interface" defined in the superclass.
      // Declare instances of the superclass, substituted by subclasses.
      Monster m1 = new FireMonster("Glumanda");   // upcast
      Monster m2 = new WaterMonster("Shiggy");  // upcast
      Monster m3 = new StoneMonster("Bisasam");  // upcast
      // Invoke the actual implementation
      System.out.println(m1.attack());  // Run FireMonster's attack()
      System.out.println(m2.attack());  // Run WaterMonster's attack()
      System.out.println(m3.attack());  // Run StoneMonster's attack()

      // m1 dies, generate a new instance and re-assign to m1.
      m1 = new StoneMonster("Boldi");  // upcast
      System.out.println(m1.attack());  // Run StoneMonster's attack()

      // We have a problem here!!!
      Monster m4 = new Monster("Mewto");
      System.out.println(m4.attack());  // garbage!!! This does not make sense
   }
}
```