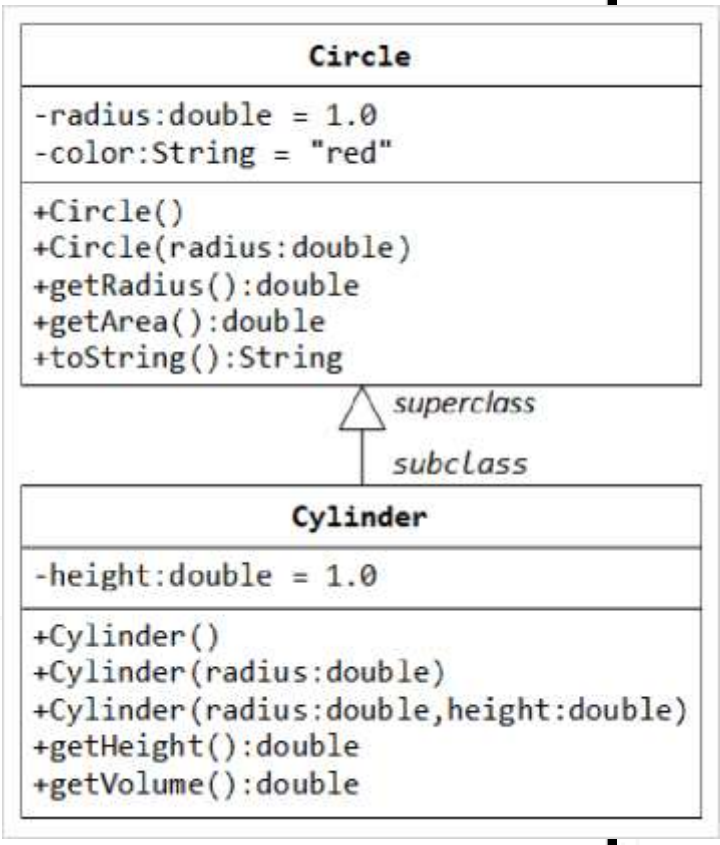
2.1 Exercise: The Circle and Cylinder Classes

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.

```
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;
   }
}
```

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 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());
   }
}
```

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.