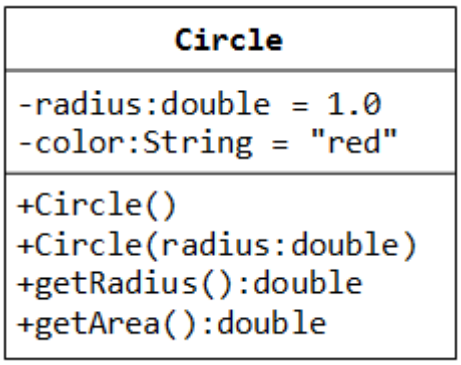
Java Practical : Revision through to inheritance

Please note that the inheritance portion starts on page 5

Also, even though you can copy/paste the code, it might be a useful exercise to type in at least some of the portions

PART 1: Revision/Catch-up practical

1.1 Exercise: The `Circle` Class



A class called **`circle`** is designed as shown in the class diagram above. It contains:

- Two `private` instance variables: `radius` (of type `double`) and `color` (of type `String`), with default value of 1.0 and "red", respectively.
- One default constructor and one *overloaded* constructor;
- Two `public` methods: `getRadius()` and `getArea()`.

The source codes for `Circle` is as follows:

```
public class Circle {           // save as "Circle.java"
   // private instance variable, not accessible from outside this class
   private double radius;
   private String color;

   // 1st constructor, which sets both radius and color to default
   public Circle() {
      radius = 1.0;
      color = "red";
   }
   // 2nd constructor with given radius, but color default
   public Circle(double r) {
      radius = r;
      color = "red";
   }
```

```
   // A public method for retrieving the radius
   public double getRadius() {
     return radius;
   }

   // A public method for computing the area of circle
   public double getArea() {
      return radius*radius*Math.PI;
   }
}
```

Can you run the `Circle` class? Why? This `Circle` class does not have a `main()` method. Hence, it cannot be run directly. This `Circle` class is a "building block" and is meant to be used in another program.

Let us write a *test program* called `TestCircle` which uses the `Circle` class, as follows:

```
public class TestCircle {        // save as "TestCircle.java"
   public static void main(String[] args) {
      // Declare and allocate an instance of class Circle called c1
      //  with default radius and color
      Circle c1 = new Circle();
      // Use the dot operator to invoke methods of instance c1.
      System.out.println("The circle has radius of "
         + c1.getRadius() + " and area of " + c1.getArea());

      // Declare and allocate an instance of class circle called c2
      //  with the given radius and default color
      Circle c2 = new Circle(2.0);
      // Use the dot operator to invoke methods of instance c2.
      System.out.println("The circle has radius of "
         + c2.getRadius() + " and area of " + c2.getArea());
   }
}
```

Now, run the `TestCircle` and study the results. Make sure you understand how this is working.

Exercises:

1. **Constructor:** Modify the class `Circle` to include a third constructor for constructing a `Circle` instance with the given `radius` and `color`.

```
// Construtor to construct a new instance of Circle with the given radius and color
public Circle (double r, String c) {......}
```

   Modify the test program `TestCircle` to construct an instance of `Circle` using this constructor.

2. **Getter:** Add a getter for variable `color` for retrieving the `color` of a `Circle` instance.

```
// Getter for instance variable color
public String getColor() {......}
```

   Modify the test program to test this method.

3. **`public` vs. `private`:** In `TestCircle`, can you access the instance variable `radius` directly (e.g., `System.out.println(c1.radius)`); or assign a new value to `radius` (e.g., `c1.radius=5.0`)? Try it out and make sure that you understand what's happening.

4. **Setter:** Is there a need to change the values of `radius` and `color` of a `Circle` instance after it is constructed? If so, add two `public` methods called *setters* for changing the `radius` and `color` of a `Circle` instance as follows:

```
// Setter for instance variable radius
public void setRadius(double r) {
   radius = r;
}

// Setter for instance variable color
public void setColor(String c) { ..... }
```

   Modify the `TestCircle` to test these methods, e.g.,

```
Circle c3 = new Circle();   // construct an instance of Circle
c3.setRadius(5.0);          // change radius
c3.setColor(...);           // change color
```

5. **Keyword "`this`":** Instead of using variable names such as `r` (for `radius`) and `c` (for `color`) in the methods' arguments, you could also use variable names `radius` (for `radius`) and `color` (for `color`) and use the special keyword "`this`" to resolve the conflict between instance variables and methods' arguments. For example,

```
// Instance variable
private double radius;


// Setter of radius
public void setRadius(double radius) {
   this.radius = radius; // "this.radius" refers to the instance variable
                         // "radius" refers to the method's argument
}
```

Modify ALL the constructors and setters in the `Circle` class to use the keyword "`this`".

6. **Method `toString()`:** Every well-designed Java class should contain a `public` method called `toString()` that returns a short description of the instance (in a return type of `String`). The `toString()` method can be called explicitly (via *`instanceName`*`.toString()`) just like any other method; or implicitly through `println()`. If an instance is passed to the `println(`*`anInstance`*`)` method, the `toString()` method of that instance will be invoked implicitly. For example, include the following `toString()` methods to the `Circle` class:

```
public String toString() {
   return "Circle: radius=" + radius + " color=" + color;
}
```

Try calling `toString()` method explicitly, just like any other method:

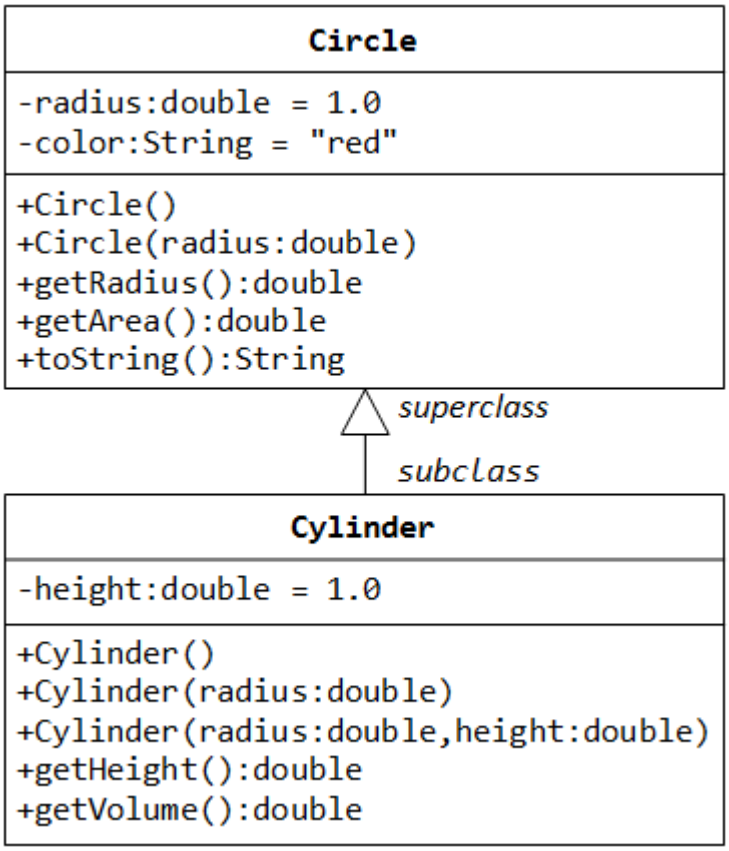
```
Circle c1 = new Circle(5.0);
System.out.println(c1.toString());   // explicit call
```

`toString()` is called implicitly when an instance is passed to `println()` method, for example,

```
Circle c2 = new Circle(1.2);
System.out.println(c2.toString());  // explicit call
System.out.println(c2); // not explicit but converted to c2.toString()
System.out.println("Operator '+' invokes toString() too: " + c2);  // '+' invokes toString() too
```

2. Exercises on Inheritance

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 (denoted by 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.java`" in the same package as Cylinder.

```
public class Cylinder extends Circle {
   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 (say, `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.

Exercise:

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).

Exercise:

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.,

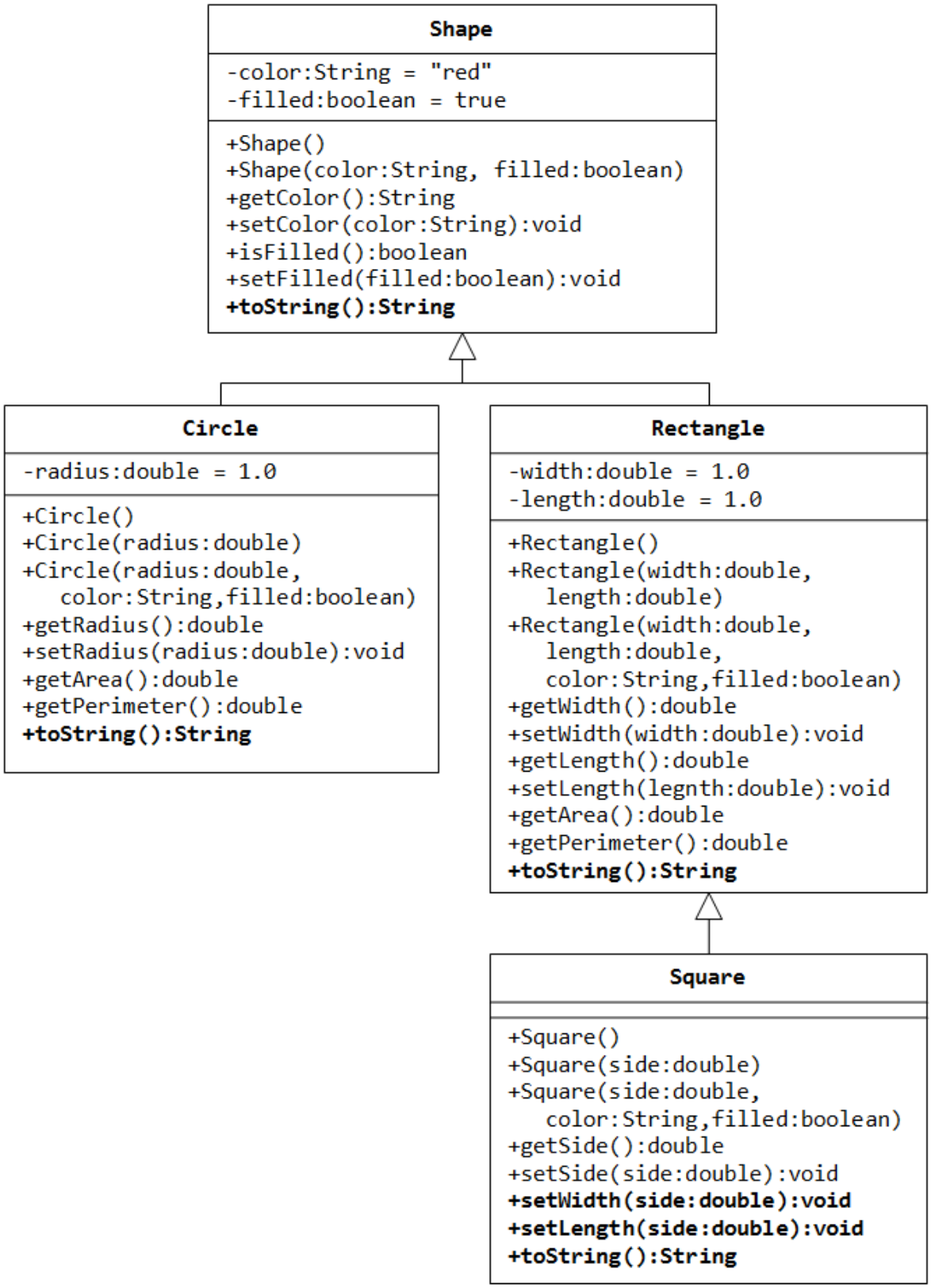
```
@Override
public String toString() {      // in Cylinder class
   return "Cylinder: subclass of " + super.toString()  // use Circle's toString()
          + " height=" + height;
}
```

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.

2.2 Exercise: Superclass Shape and its subclasses Circle, Rectangle and Square

Note: Since this is a different set of classes you should create them in a new package.



1. 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`".

2. Write a test program to test all the methods defined in `Shape`.

3. 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.

4. 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.

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

```
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.