

Abstract Classes and Interfaces

Abstract Classes

Why Abstract Methods

Interesting Points on Abstract Classes

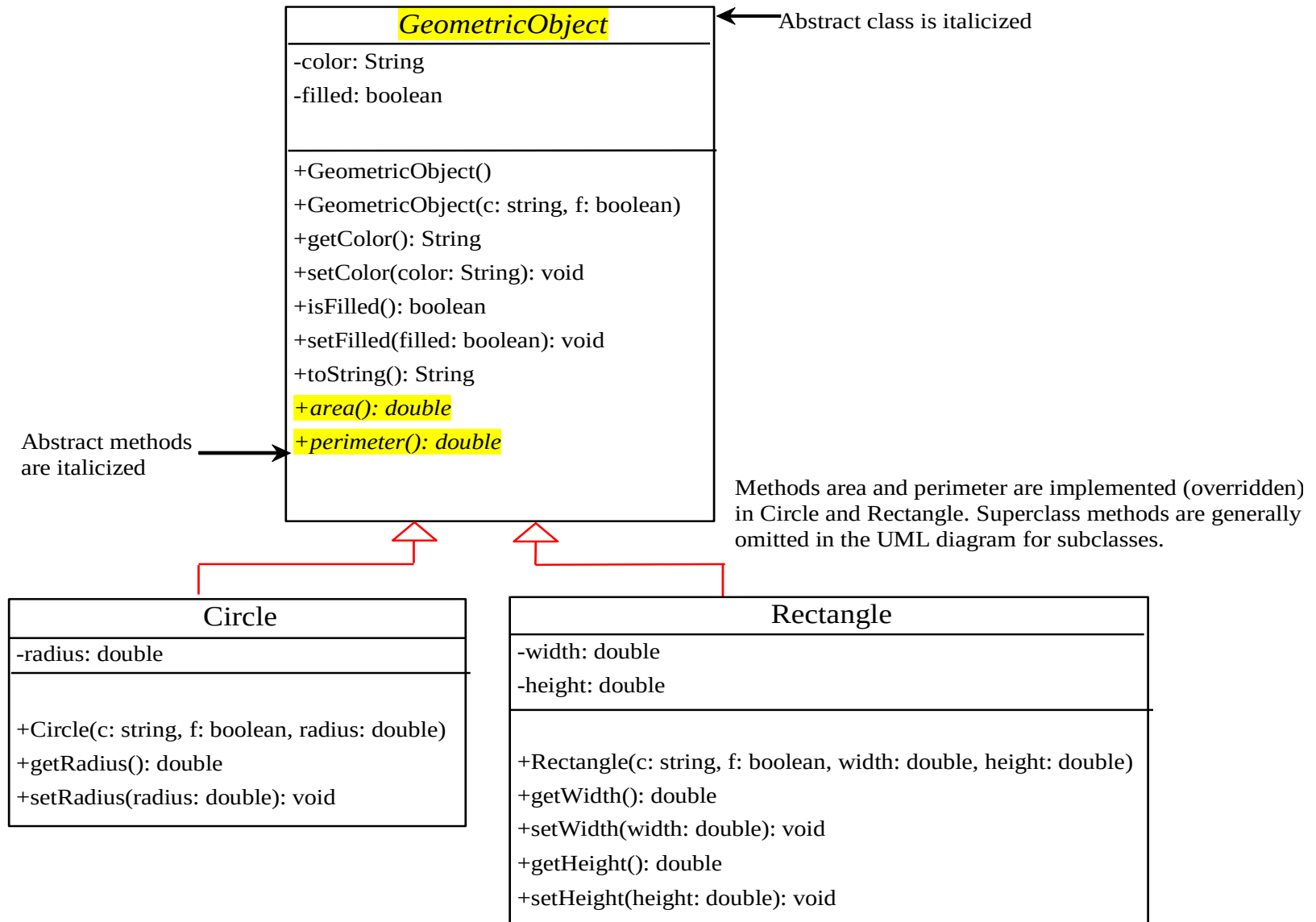
Abstract Classes

- An abstract class cannot be used to create objects. So if the class **GeometricObject** is abstract then writing the statement below will cause a syntax error:

```
GeometricObject g1 = new GeometricObject("Red", true);
```

- Abstract classes are denoted using the abstract modifier in the class header.
- An abstract class can contain abstract methods, which are implemented in concrete (non abstract) subclasses.
- In UML notation, the names of abstract classes and their abstract methods are italicized.

Abstract Classes and Abstract Methods



Abstract Class GeometricObject

```
public abstract class GeometricObject {  
    private String color ;  
    private boolean filled;  
    public GeometricObject() { // No arg constructor  
        color = "white";  
        filled = false;  
    }  
    public GeometricObject(String c, boolean f) {  
        color = c;  
        filled = f;  
    }  
    public String getColor() {  
        return color;  
    }  
    public void setColor(String c) {  
        color = c;  
    }  
    public boolean isFilled() {  
        return filled;  
    }  
    public void setFilled(boolean f) {  
        filled = f;  
    }  
}
```

Note the use of the keyword **abstract** in the class header and the methods headers.



```
    public String toString() {  
        String s = "The color is " + color;  
        if (filled)  
            s+= ".\nThe geometric object is filled";  
        else  
            s+= ".\nThe geometric object is not filled";  
        return s;  
    }  
  
    // Abstract methods do not have a body  
    public abstract double area();  
    public abstract double perimeter();  
}  
// end class GeometricObject
```

Class Circle

```
public class Circle extends GeometricObject {
    private double radius;
    public Circle(String c, boolean f, double r) {
        super(c,f); // calls constructor of GeometricObject using the keyword super
        this.radius = r;
    }
    public double getRadius() {
        return radius;
    }
    public void setRadius(double radius) {
        this.radius = radius;
    }
    public double area() {
        return radius * radius * Math.PI;
    }
    public double perimeter() {
        return 2 * radius * Math.PI;
    }
    public String toString() {
        // call toString of GeometricObject using the keyword super
        return super.toString() + ".\nThe radius is " + radius;
    }
} // end class Circle
```

Class Rectangle

```
public class Rectangle extends GeometricObject {
    private double width;
    private double height;
    public Rectangle(String c, boolean f, double width, double height) {
        super(c, f); // calls constructor of GeometricObject
        this.width = width;
        this.height = height;
    }
    public double getWidth() {
        return width;
    }
    public void setWidth(double width) {
        this.width = width;
    }
    public double getHeight() {
        return height;
    }
    public void setHeight(double height) {
        this.height = height;
    }
    public double area() {
        return width * height;
    }
    public double perimeter() {
        return 2 * (width + height);
    }
    public String toString() {
        // call toString of GeometricObject
        return super.toString() +
            ".\nThe height is " + height +
            ".\nThe width is " + width;
    }
} // end class Rectangle
```



Why Abstract Methods

- You may be wondering what advantage is gained by defining the methods **area** and **perimeter** as abstract in the `GeometricObject` class.
- Let us revisit the example of polymorphism (chapter 11):

```
GeometricObject [] g = new GeometricObject[120];
```

We can fill this array with `Circle` and `Rectangle` Objects only. It is not possible to store in it objects of the type `GeometricObject` since it is abstract. So we can write:

```
g[0] = new Circle("Red", true, 1); // g[0] refers to a Circle object
```

```
g[1] = new Rectangle("Red", true, 5, 6); // g[1] refers to a Rectangle
```

```
...
```

```
g[119] = new _____; // g[119] can refer to a Circle or Rectangle
```

Why Abstract Methods (Continued)

To print the area of Circle and Rectangle Objects we can write:

```
for (int i = 0; i < g.length; i++)  
    System.out.print(g[i].area());
```

If area() was not included as an abstract method in the class GeometricObject then we would have to write more code to perform the same task above as follows:

```
for (int i = 0; i < g.length; i++)  
    if (g[i] instanceof Circle)  
        System.out.print(((Circle)g[i]).area());  
    else if (g[i] instanceof Rectangle)  
        System.out.print(((Rectangle)g[i]).area());
```

The more subclasses of the class GeometricObject there are, the more the if statements we have to add.

Why Abstract Methods (Continued)

So the advantage of having the abstract methods `area()` and `perimeter()` in the superclass `GeometricObject` is to benefit from the idea of polymorphic call and dynamic binding to make the code simpler.

Note that the methods `area()` and `perimeter()` cannot be implemented (have their code written) in the `GeometricObject` class, because their implementation (code) depends on the specific type of geometric object. This means that we know how to compute the area of a circle and that of a rectangle, but there is no formula to compute the area of a geometric object because it does not exist as an entity by itself in real life. It is an abstract concept and that is why the class `GeometricObject` is made abstract.

Such methods are referred to as *abstract methods* and are denoted using the **abstract** modifier in the method header. They have no body, not even curly brackets `{ }`, and the method header is followed by a semicolon.

Interesting Points on Abstract Classes

1. Abstract classes are like regular classes, but you cannot create instances of abstract classes using the **new** operator. Hence, the following statement is not allowed:

```
GeometricObject go1 = new GeometricObject("White",true);
```

2. An abstract class can be used as a data type. Therefore, the following statement, which creates an array whose elements are of the **GeometricObject** type, is correct.

```
GeometricObject[] gArray = new GeometricObject[10];
```

You can then create an instance of **Circle** and store it in the array like this:

```
gArray[0] = new Circle();
```

3. You can still define constructors for an abstract class, which are invoked in the constructors of its subclasses. For instance, the constructors of **GeometricObject** are invoked in the **Circle** class and the **Rectangle** class.

Interesting Points on Abstract Classes (Continued)

4. An abstract method is defined without implementation. Its implementation is provided by the subclasses. If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined abstract. In other words, in a nonabstract (concrete) subclass extended from an abstract class, all the abstract methods must be implemented.
5. A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that contains no abstract methods.
6. An abstract method cannot be contained in a nonabstract (concrete) class.