Abstract Classes in java

Java Abstract classes are used to declare common characteristics of subclasses. **An abstract class cannot be instantiated. It can only be used as a superclass for other classes that extend the abstract class.** Abstract classes are declared with the abstract keyword. Abstract classes are used to provide a template or design for **concrete** subclasses down the inheritance tree.

Like any other class, an abstract class can contain fields that describe the characteristics and methods that describe the actions that a class can perform. **An abstract class can include methods that contain no implementation. These are called abstract methods. The abstract method declaration must then end with a semicolon rather than a block.** If a class has any abstract methods, whether declared or inherited, the entire class must be declared abstract. Abstract methods are used to provide a template for the classes that inherit the abstract methods.

Abstract classes cannot be instantiated; they must be subclassed, and actual implementations must be provided for the abstract methods. Any implementation specified can, of course, be overridden by additional subclasses. An object must have an implementation for all of its methods. You need to create a subclass that provides an implementation for the abstract method.

A class Vehicle might be specified as abstract to represent the general abstraction of a vehicle, as creating instances of the class would not be meaningful.

```
abstract class Vehicle {

	int numofGears;
	String color;
	abstract boolean hasDiskBrake();
	abstract int getNoofGears();
}
```

Example of a shape class as an abstract class

```
//Here I am declaring an abstract class
//Abstract classes usually have at least one abstract method
//In this case i'm making getArea() abstract because I want
//to make sure that each subclass overrides (i.e. provides)
//this method.
//Check out the syntax for an abstract method
public abstract class Shape
{
  private String color;
   private boolean filled;

   public Shape()
   {
      color = "red";
      filled = true;
   }

   public Shape(String color, boolean filled)
   {
      this.color = color;
      this.filled = filled;
   }

   public String getColor()
   {
      return color;
   }

   //Note the use of "is" instead of "get" for boolean fields
   public boolean isFilled()
   {
      return filled;
   }

  public void setColor(String c)
   {
     color = c;
   }

   public void setFilled(boolean f)
   {
      filled = f;
   }


   //Look at the empty abstract method
   //and note that use of the semicolon instead
   //of the curly braces
   public abstract double getArea();
```

```
    public String toString()
    {
        return "Shape: color=" + color + " filled=" + filled;
    }
}
```

Above, we implement the generic Shape class as an abstract class so that we can draw lines, circles, triangles etc. All shapes have some common fields and methods, but each can, of course, add more fields and methods. The abstract class guarantees that each shape will have the same set of basic properties. We declare this class abstract because there is no such thing as a generic shape. There can only be concrete shapes such as squares, circles, triangles etc.

The important piece of code above is public abstract double getArea(); because it forces all subclasses to override this method.

Now, my Circle class looks like this:

```
public class Circle extends Shape
{
    private double radius;

    public Circle()
    {
        super();
        radius = 1.0;
    }

    public Circle(double r)
    {
        super();
        radius = r;
    }

    public Circle(double r, String c, boolean f)
    {
        super(c,f);
        radius = r;
    }
```

```
    ////////////////////////// End of constructors ////////////

    public double getRadius()
    {
       return radius;
    }

    public void setRadius(double r)
    {
       radius = r;
    }


    //NB: We MUST override getArea - Why?
    //Comment out the method to see what the compiler has to say
    public double getArea()
    {
       return radius*radius*Math.PI;
    }

    //We didn't declare this as abstract in our superclass
    //so subclasses can choose whether they want to implement it or not
    public double getPerimeter()
    {
      return 2*radius*Math.PI;
    }

    public String toString()
    {
       return "Circle: subclass of " + super.toString()
             + " radius=" + radius;

    }
}
```

Notes:

The subclass must define an implementation for every abstract method of the abstract superclass, **or the subclass itself will also be abstract** (and must be declared as such.)

One potential disadvantage of using abstract classes (and, indeed, ordinary classes) is not being able to use multiple inheritance. In the sense, when a class extends an abstract class, it can't extend any other class.