**Topic:** Abstract Classes.
**OOP concepts involved:** Inheritance, Abstract Classes, Class Instantiation and Declaration, Class Members, Polymorphism, Constructor Methods, Class Modifiers, Getters and Setters.
**Programming generic concepts involved:** Access Modifiers, Functions, Primitive Variables, Variable Declaration, Function Parameters.

---

## ➢ Theoric introduction

### WHAT IS ABSTRACTION?

Abstraction is a process of hiding the implementation details from the user. Only the functionality will be provided to the user. In Java, abstraction is achieved using abstract classes and interfaces. Abstraction is one of the four major concepts behind object-oriented programming (OOP).

### ABSTRACT CLASSES

An abstract class is a class that is declared *abstract* (it may or may not include abstract methods). Abstract classes cannot be instantiated, but they can be subclassed.

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY); //Abstract method which
receives 2 parameters of type double and returns void
```

If a class includes abstract methods, then the class itself must be declared *abstract*, as in:

```
public abstract class GraphicObject {
   // Members Variables (Attributes/Fields) declaration
   // Non-Abstract methods declaration
```

```
    abstract void draw();
}
```

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared *abstract*.

It can have abstract methods(methods without body) as well as concrete methods (regular methods with body). A normal class(non-abstract class) cannot have abstract methods.

### WHY WOULD I NEED ABSTRACT CLASSES?

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method, because everyone of the  subclasses need to have the same behavior but with a different implementation.

### ABSTRACT CLASSES COMPARED TO INTERFACES

Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation. However, with abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods. With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public. In addition, you can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces.

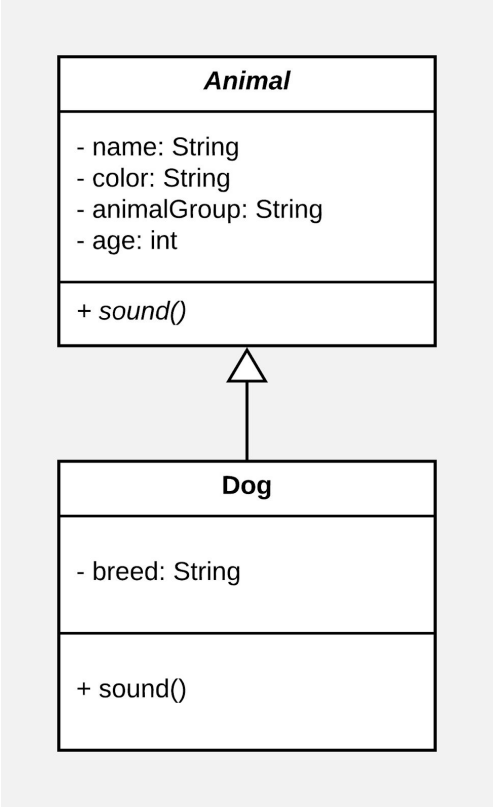
**Which should you use, abstract classes or interfaces?**

- ❏ Consider using abstract classes if any of these statements apply to your situation:
    - ➔ You want to share code among several closely related classes.
    - ➔ You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
    - ➔ You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.
- ❏ Consider using interfaces if any of these statements apply to your situation:
    - ➔ You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
    - ➔ You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
    - ➔ You want to take advantage of multiple inheritance of type.

➢ **Statement**

Make use of the principle of abstraction and create a simple class relationship between a parent Animal Class and a child Dog Class. Make sure of using Abstract Classes.

## ➢ Class design (UML)




## ➢ Program Code

Animal.java

```
public abstract class Animal {

	// Attributes declaration
	private String name;
	private String color;
	private String animalGroup;
	private int age;

	// Constructors
	public Animal() {}

	public Animal(String name, String color, String animalGroup, int age) {
		super();
		this.name = name;
		this.color = color;
```

```
		this.animalGroup = animalGroup;
		this.age = age;
	}

	// Abstract method
	public abstract void sound();

	// Getters and setters
	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getColor() {
		return color;
	}

	public void setColor(String color) {
		this.color = color;
	}

	public String getAnimalGroup() {
		return animalGroup;
	}

	public void setAnimalGroup(String animalGroup) {
		this.animalGroup = animalGroup;
	}

	public int getAge() {
		return age;
	}

	public void setAge(int age) {
		this.age = age;
	}
}
```

Dog.java

```
public class Dog extends Animal{

	// Declaring its own attributes
```

```
	private String breed;

	// Constructors
	public Dog(String name, String color, String animalGroup, int age, String
breed) {
		super(name, color, animalGroup, age);
		this.breed = breed;
	}

	// Adding our own implementation of the parent abstract method
	@Override
	public void sound() {
		System.out.println("Woof Woof!");
	}

	// Getters and setters
	public String getBreed() {
		return breed;
	}

	public void setBreed(String breed) {
		this.breed = breed;
	}
}
```

Main.java

```
public class Main {
	public static void main(String[] args) {
		// Instantiating Dog Object
		Dog doggy = new Dog("Rusty", "Black", "Mammal", 6, "Rottweiler");
		// Executing dog's behavior
		System.out.println(doggy.getName() + " is a " + doggy.getAge() + "
year old " + doggy.getColor() + " " + doggy.getBreed() + " and when it makes a
sound, he usually bark:");
		doggy.sound();
	}
}
```

➢ **Program execution**

This simple program example captures data related to the Animal Class attributes. After data is captured, its name, age, color, breed are showed within the command line and then, Dog Class *sound()* method is called showing its own Dog implementation from the abstract parent method.

```
Rusty is a 6 year old Black Rottweiler and when it makes a sound, he usually bark:
Woof Woof!
```

➢ **Conclusions**

It is convenient to use Abstract Classes when you want to have the same behavior on different subclasses using a different implementation within each one of them. It is a good practice due to the code recycling part, when you need to use the same members as the parent class inside your different subclasses.

Abstract methods only have the definition of the behavior of Class, in the other hand, subclasses provide the specific behavior implementation.