What is an Abstract Class?

- **Abstract Class**: An abstract class in Java is a class that cannot be instantiated directly. It is used to define a base class that other subclasses can extend. An abstract class can have both **abstract methods** (methods without implementation) and **concrete methods** (methods with implementation).
- **Abstract Method**: An abstract method is a method that is declared without a body, and the implementation of the method is provided by subclasses that extend the abstract class

NOTE

1. **Abstract classes** cannot be instantiated directly. They are meant to be extended by other classes.
2. **Abstract methods** in an abstract class must be implemented by the first concrete (non-abstract) subclass.
3. A **concrete method** in an abstract class can have a method body and can be called by subclasses.

Syntax for Abstract Class and Method in Java:

```
abstract class ClassName {
    // Abstract method
    public abstract void methodName();

    // Concrete method
    public void concreteMethod() {
        System.out.println("This is a concrete method.");
    }
}
```

Example Program:

Let's go step by step through a program that demonstrates the use of abstract classes and methods in Java.

```
// Define an abstract class Animal
abstract class Animal {
    // Abstract method (does not have a body)
    public abstract void sound();

    // Concrete method with a body
    public void breathe() {
        System.out.println("Breathing...");
    }
}

// Define a Dog class that extends Animal
class Dog extends Animal {
    // Implement the abstract method sound
    public void sound() {
        System.out.println("Woof!");
    }
}

// Define a Cat class that extends Animal
class Cat extends Animal {
    // Implement the abstract method sound
    public void sound() {
        System.out.println("Meow!");
    }
}

public class Main {
    public static void main(String[] args) {
        // Cannot instantiate the abstract class Animal directly
        // Animal animal = new Animal(); // This will give an error

        // Instantiate the concrete subclasses
        Dog dog = new Dog();
        Cat cat = new Cat();

        // Call the methods on the Dog and Cat objects
        dog.sound();    // Outputs: Woof!
        dog.breathe();  // Outputs: Breathing...

        cat.sound();    // Outputs: Meow!
        cat.breathe();  // Outputs: Breathing...
    }
}
```

Explanation of the Program:

1. **Abstract Class `Animal`**:
   - We define an abstract class `Animal`.
   - It has one **abstract method** `sound()` with no implementation. This method is meant to be implemented by subclasses.
   - It has one **concrete method** `breathe()`, which prints "Breathing...". This method does not need to be implemented by subclasses, and they can call it directly.
2. **Concrete Subclasses**:
   - The `Dog` and `Cat` classes are **concrete** subclasses of the `Animal` class. They **must implement** the `sound()` method, as it is abstract in the parent class `Animal`.
   - Both `Dog` and `Cat` classes provide their specific implementations of the `sound()` method:
     - `Dog` class prints "Woof!".
     - `Cat` class prints "Meow!".
3. **In the `Main` Class**:
   - **Instantiating subclasses**: We create instances of `Dog` and `Cat` because an abstract class like `Animal` cannot be instantiated directly.
   - We call the `sound()` method on both the `Dog` and `Cat` objects to demonstrate polymorphism.
   - We also call the `breathe()` method, which is inherited from the `Animal` class.

Output of the Program:

```
Woof!
Breathing...
Meow!
Breathing...
```

- **Abstract Methods**: Define the signature of methods that must be implemented by subclasses.
- **Concrete Methods**: Can be provided in the abstract class and can be inherited by subclasses.
- **Polymorphism**: In the example, both `Dog` and `Cat` provide their own implementation of the abstract `sound()` method, demonstrating polymorphism where the same method call (`sound()`) behaves differently depending on the object type.

Why Use Abstract Classes?

- **Code Reusability**: Abstract classes allow you to define common methods that can be shared by all subclasses (like `breathe()` in the example).
- **Enforcing Method Implementation**: Abstract methods ensure that every subclass will have to provide its own specific implementation for the abstract methods.