**Abstract Class**

**Purpose:**

- An abstract class is used when you want to create a base class that defines a common structure or behavior for its subclasses but should not be instantiated on its own.

- It allows you to define methods that must be implemented by subclasses, as well as methods that are already implemented.

**Key Features:**

- Abstract Methods: Can include abstract methods (methods without a body) that must be implemented by the subclass.

- Concrete Methods: Can also include concrete methods (methods with a body).

- Instance Variables: Can have instance variables (fields).

- Constructor: Can have a constructor, which is useful for initializing fields.

- Single Inheritance: A class can only inherit from one abstract class due to Java's single inheritance model.

When to Use:

- Use an abstract class when:

- You want to provide a common base with shared code (e.g., methods, fields) to related classes.

- You have default behavior that can be shared by multiple subclasses but still allow them to override or add to this behavior.

- You need to maintain state (i.e., instance variables) that should be inherited by subclasses.

Example:

java

abstract class Animal {

String name;

Animal(String name) {

this.name = name;

}

abstract void sound(); // Abstract method

void sleep() { // Concrete method

System.out.println(name + " is sleeping");

}

}

class Dog extends Animal {

Dog(String name) {

super(name);

}

void sound() {

System.out.println(name + " says Woof");

}

}

2. **Interface**

Purpose:

- An interface is used to define a contract that a class must adhere to. It specifies what methods a class must implement but not how they are implemented.

- It allows multiple classes, possibly unrelated, to implement the same set of methods, promoting a flexible design.

Key Features:

- Abstract Methods Only: Originally, interfaces could only contain abstract methods (methods without a body), but with Java 8 and later, interfaces can also contain default methods (with a body) and static methods.

- No Instance Variables: Cannot have instance variables (except static final constants).

- Multiple Inheritance: A class can implement multiple interfaces, allowing for more flexible design.

- No Constructor:Cannot have a constructor since interfaces are not classes.

When to Use:

- Use an interface when:

- You want to define a role that can be shared across different classes (e.g., both classes Dog and Car could implement a Moveable interface).

- You want to achieve multiple inheritance in Java by allowing a class to implement multiple interfaces.

- You need to define behavior that can be implemented differently by unrelated classes.

Example

java

interface Flyable {

void fly(); // Abstract method

}

interface Swimmable {

void swim(); // Abstract method

}

class Duck implements Flyable, Swimmable {

public void fly() {

System.out.println("Duck is flying");

}

public void swim() {

System.out.println("Duck is swimming");

}

}

When to Use Abstraction vs. Interface

- Abstract Class

- When you have a clear inheritance hierarchy (e.g., Animal -> Dog, Cat).

- When you need to share code (methods/fields) between closely related classes.

- When you want to partially implement some behavior and leave the rest to subclasses.

- Interface:

- When you need to define a common set of behaviors that can be implemented by unrelated classes (e.g., Flyable, Swimmable).

- When you need to achieve multiple inheritance (e.g., a class can implement multiple interfaces).

- When you are more focused on defining a contract rather than providing a common implementation.

Real-World Example:

- Abstract Class Example:

- You might have an abstract class Vehicle with methods like startEngine() and stopEngine(), which are common to all vehicles, but leave the implementation of methods like drive() to the subclasses Car, Bike, etc.

- Interface Example:

- You might define an interface Payable that has a method calculatePay(), which can be implemented by different classes like Employee, Contractor, and Freelancer. These classes might have very different logic for how they calculate pay, but they all adhere to the Payable contract.

Understanding the use cases for abstract classes and interfaces will help you decide which to use based on the design and requirements of your Java programs.