**1. Abstract Class**

An **abstract class** allows you to define both **abstract methods** (methods without implementation) and **concrete methods** (methods with implementation). It provides a way to enforce a common contract while also allowing some implementation details to be shared across subclasses.

* **When to use:** When you want to provide some shared behavior and leave other parts to be implemented by subclasses. It’s also used when you want to allow certain methods to have default behavior but force others to be implemented by the subclasses.

**Example:**

abstract class Animal {

abstract void sound(); // Abstract method, needs to be implemented by subclasses

void breathe() { // Concrete method

System.out.println("Breathing...");

}

}

**2. Interface**

An **interface** defines a contract with only **abstract methods** (until Java 8, after which default and static methods were introduced). It does not provide any implementation at all (unless using default or static methods in newer Java versions). All methods in an interface are implicitly abstract (unless they're static or default). Classes that implement an interface must provide implementations for the abstract methods.

* **When to use:** When you want to define a **contract** that any implementing class must follow. Interfaces are great for **polymorphism**, especially when you need to apply the same behavior across multiple class hierarchies (since Java allows a class to implement multiple interfaces).

**Example:**

interface Animal {

void sound(); // Abstract method to be implemented

}

**3. Non-Abstract Class (Concrete Class)**

Even in a **non-abstract class**, you can achieve abstraction by **hiding the implementation details** through **encapsulation**. This is done by using **private fields** and providing public getter/setter methods or public business logic methods that encapsulate complexity. The users of the class interact with the object through these methods, without needing to know how the internal data is being managed.

* **When to use:** When you want to provide a fully implemented class but still want to **hide implementation details** from the users of the class. This is done by providing a **simple interface** and keeping the **internal workings** private.

**Example:**

public class BankAccount {

private double balance;

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

}

}

public void withdraw(double amount) {

if (amount <= balance) {

balance -= amount;

}

}

public double getBalance() {

return balance;

}

}

* **Abstraction** can be achieved in Java using:
  + **Abstract classes** (to define common behavior with some methods having default implementations and others being abstract).
  + **Interfaces** (to define a contract for the behavior that any class must follow, with no implementation until Java 8+).
  + **Non-abstract (concrete) classes** (to **encapsulate complexity** and provide a simplified interface to the users).

Each approach provides a different way to hide complexity and expose only relevant details, so the best approach depends on the use case.