If we consider an **ATM (Automated Teller Machine)** as a **real-life example of abstraction**, it actually provides **both Data Abstraction and Control Abstraction**, but in different ways.

**1️ Data Abstraction in ATM (Hiding Data Implementation)**

**What it hides?**

* The ATM **hides the internal details of data storage**, such as how your bank balance, transactions, and account details are **stored in the bank’s database**.
* You **only see your balance or transaction details**, but **not how they are retrieved, stored, or calculated**.

**Example:**

* When you **check your balance**, you don’t know:
  + Where the balance is stored.
  + How it is fetched from the database.
  + How many tables are involved in the backend database.

**How is this related to Java?**

* Similar to an **abstract class or interface** that defines behavior but hides implementation.

**Java Example for Data Abstraction (Like ATM Balance Check)**

// Abstract class providing data abstraction

abstract class ATM {

abstract double getBalance(); // Hides how balance is retrieved

}

// Concrete implementation

class UserATM extends ATM {

private double balance = 5000; // Hidden data

@Override

double getBalance() {

return balance; // User only gets the balance, not how it's stored

}

}

public class ATMDataAbstraction {

public static void main(String[] args) {

ATM myATM = new UserATM();

System.out.println("Your balance: " + myATM.getBalance()); // Only balance is exposed

}

}

**Imp note:**

* **Data Abstraction in an ATM hides how data (balance) is stored & retrieved**—the user **only sees the result.**

**2️ Control Abstraction in ATM (Hiding Control Flow)**

**What it hides?**

* The ATM **hides the step-by-step process** of handling a transaction.
* When you **withdraw cash**, you don’t know:
  + How the ATM **verifies your PIN** internally.
  + How the bank **checks if your account has enough balance**.
  + How the bank **updates your account after withdrawal**.

**Example:**

* You **press a button** to withdraw money, and it just works.
* Internally, multiple checks and calculations happen, but you don’t need to worry about them.

**How is this related to Java?**

* Similar to **method calls** in Java, where the **complex logic is hidden inside a function**.

**Java Example for Control Abstraction (Like ATM Withdrawal Process)**

class ATM {

// Control Abstraction: User calls withdraw() without knowing the internal process

public void withdrawMoney(double amount) {

if (authenticateUser() && checkBalance(amount)) {

dispenseCash(amount);

updateBalance(amount);

System.out.println("Transaction successful!");

} else {

System.out.println("Transaction failed!");

}

}

// Internal steps (hidden from user)

private boolean authenticateUser() {

System.out.println("User authenticated.");

return true; // Simulating successful authentication

}

private boolean checkBalance(double amount) {

System.out.println("Checking balance...");

return amount <= 5000; // Simulating balance check

}

private void dispenseCash(double amount) {

System.out.println("Dispensing " + amount + " cash.");

}

private void updateBalance(double amount) {

System.out.println("Updating balance in the database.");

}

}

public class ATMControlAbstraction {

public static void main(String[] args) {

ATM myATM = new ATM();

myATM.withdrawMoney(1000); // User only calls withdrawMoney(), rest is hidden

}

}

**Imp note :**

* **Control Abstraction in an ATM hides the complex steps behind an action (withdrawal)**—the user **only presses a button and gets money**.