Southeast University

Department of Computer Science and Engineering (CSE)
School of Sciences and Engineering

Semester: (Summer, Year: 2025)

LAB REPORT NO: **08**
Course Title: Introduction to Programming Language II (Java) Lab
Course Code: CSE282.2
Batch: 65

Lab Experiment Name: Object Oriented Programming Concepts (Abstraction).

Student Details

	Name	ID
1.	Md. Mahmud Hossain	2023200000799

Submission Date : 19-09-25
Course Teacher's Name : Dr. Mohammed Ashikur Rahman

Lab Report Status	
Marks: …………………………………	**Signature:....................**
Comments:...	**Date:.............................**

Lab Task 8 : Object Oriented Programming Concepts (Abstraction) Problems:

2. Write a program to manage different types of vehicles, such as cars, motorcycles, and trucks. Each vehicle type requires common attributes defined in an abstract class (e.g., model, year), type-specific properties, interfaces for behavior (e.g., startEngine()), and multiple inheritance to handle vehicle-specific interfaces.

3. Write a program to manage different types of bank accounts, such as savings, checking, and credit. Each account type requires common attributes defined in an abstract class (e.g., accountNumber, balance), type-specific properties, interfaces for behavior (e.g., deposit(), withdraw()), and multiple inheritance to handle account-specific interfaces.

Vehicle Management System using Abstract Classes and Interfaces in Java Objective

The objective of this program is to design a vehicle management system that demonstrates the application of **abstraction, interfaces, inheritance, and polymorphism**. The system models different types of vehicles—**Car, Motorcycle, and Truck**—each with unique attributes and functionalities such as engine starting and load capacity handling.

Problem Analysis

Vehicles share common properties like **model** and **year**, but they differ in behavior and specifications. Cars have doors, motorcycles may have sidecars, and trucks handle load capacities. To address this, the abstract class Vehicle defines the base attributes and enforces the method displayInfo(). The Engine interface provides a common behavior for starting engines, while the Loadable interface introduces additional functionality for trucks. By combining abstract classes and interfaces, the system can manage multiple vehicle types with flexible, reusable, and extensible code.

Background Theory

- **Abstract Class (Vehicle)**: Encapsulates shared attributes (model, year) and declares the abstract method displayInfo() for subclasses to implement.
- **Interfaces (Engine, Loadable)**: Provide contracts for common behaviors. Engine ensures all vehicles can start their engines, while Loadable applies only to trucks for load handling.
- **Inheritance**: Car, Motorcycle, and Truck extend Vehicle, inheriting base properties.
- **Polymorphism**: Different vehicles are referenced as Vehicle or interfaces (Engine, Loadable) but exhibit unique behaviors at runtime.
- **Method Overriding**: Subclasses implement their own versions of displayInfo() and startEngine().

Algorithm Design

1. **Define Abstract Class – Vehicle** ○ Attributes: model, year.
    - Abstract method: displayInfo().
2. **Define Interfaces** ○ Engine: method startEngine().
    - Loadable: method loadCapacity(int capacity).
3. **Implement Subclasses**
    - **Car**: adds attribute numberOfDoors, overrides displayInfo(), implements startEngine().
    - **Motorcycle**: adds attribute hasSidecar, overrides displayInfo(), implements startEngine(). ○ **Truck**: adds attribute maxLoad, overrides displayInfo(), implements both startEngine() and loadCapacity().
4. **Main Class – VehicleDemo** ○ Create objects of Car, Motorcycle, and Truck using **upcasting** to Vehicle.
    - Call displayInfo() polymorphically. ○ Use **interface casting** ((Engine), (Loadable)) to call specialized methods.

Conclusion

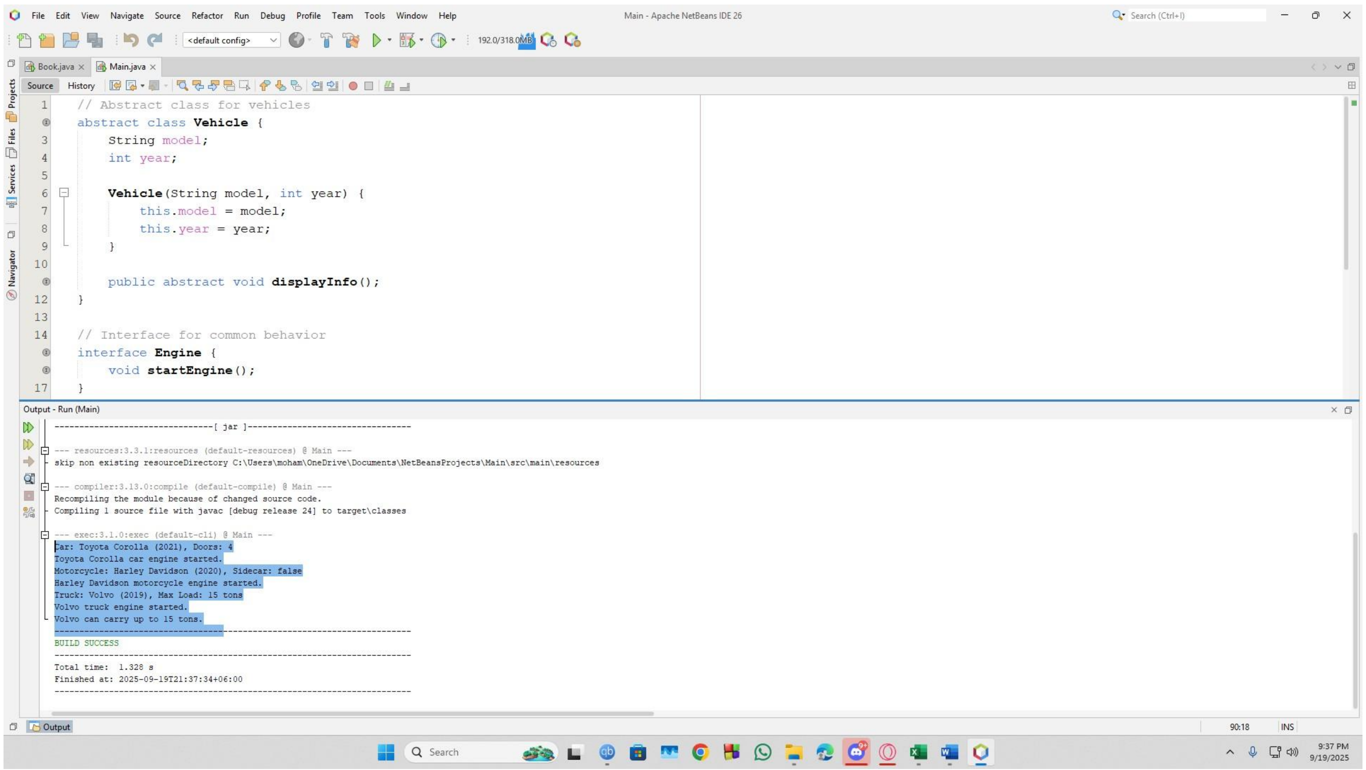
This Vehicle Management System effectively demonstrates how **abstract classes and interfaces** can be used to model real-world entities with shared and unique behaviors. By separating common attributes in the Vehicle abstract class and defining additional behaviors in interfaces, the program achieves **code reusability, extensibility, and flexibility**. Cars, motorcycles, and trucks are managed in a unified framework while still preserving their unique characteristics.

```
// Abstract class for vehicles
abstract class Vehicle {
String model;
   int year;

   Vehicle(String model, int year)
{        this.model = model;
this.year = year;
   }

   public abstract void displayInfo();
}
```

```
// Interface for common behavior
interface Engine {     void
startEngine();
}

// Interface for load capacity (specific to trucks)
interface Loadable {     void
loadCapacity(int capacity);
}

// Car class
class Car extends Vehicle implements Engine {
int numberOfDoors;

    Car(String model, int year, int numberOfDoors) {
        super(model, year);
this.numberOfDoors = numberOfDoors;
    }
    @Override
    public void startEngine() {
        System.out.println(model + " car engine started.");
    }
    @Override
    public void displayInfo() {
        System.out.println("Car: " + model + " (" + year + "), Doors: " + numberOfDoors);
    }
}

// Motorcycle class
class Motorcycle extends Vehicle implements Engine {
boolean hasSidecar;

    Motorcycle(String model, int year, boolean hasSidecar) {
super(model, year);         this.hasSidecar = hasSidecar;
    }
    @Override
    public void startEngine() {
        System.out.println(model + " motorcycle engine started.");
    }
    @Override
```

```
    public void displayInfo() {
        System.out.println("Motorcycle: " + model + " (" + year + "), Sidecar: " + hasSidecar);
    }
}

// Truck class
class Truck extends Vehicle implements Engine, Loadable {
    int maxLoad;

    Truck(String model, int year, int maxLoad) {
        super(model, year);
        this.maxLoad = maxLoad;
    }
    @Override
    public void startEngine() {
        System.out.println(model + " truck engine started.");
    }
    @Override
    public void loadCapacity(int capacity) {
        System.out.println(model + " can carry up to " + capacity + " tons.");
    }
    @Override
    public void displayInfo() {
        System.out.println("Truck: " + model + " (" + year + "), Max Load: " + maxLoad + " tons");
    }
}

// Main class
public class VehicleDemo {     public
static void main(String[] args) {
        Vehicle v1 = new Car("Toyota Corolla", 2021, 4);
        Vehicle v2 = new Motorcycle("Harley Davidson", 2020, false);
        Vehicle v3 = new Truck("Volvo", 2019, 15);

        v1.displayInfo();
        ((Engine) v1).startEngine();

        v2.displayInfo();
        ((Engine) v2).startEngine();
```

```
        v3.displayInfo();
        ((Engine) v3).startEngine();
        ((Loadable) v3).loadCapacity(15);
    }
}
```



Bank Account Management System using Abstract Classes and Interfaces in Java Objectve

The objective of this program is to implement a simple banking system that demonstrates the use of **abstract classes, interfaces, inheritance, polymorphism, and method overriding** in Java. The system manages different types of accounts—**Savings, Checking, and Credit**—each with distinct rules for deposits and withdrawals, while sharing common properties such as account number and balance.

Problem Analysis

In a banking system, different accounts share some similarities (account number, balance, and type) but also differ in behavior. Savings accounts offer interest and prevent overdrafts, checking accounts allow overdrafts, and credit accounts include a credit limit. To model this scenario, the abstract class BankAccount defines common attributes and an abstract method accountType(). The Transaction interface enforces deposit and withdrawal operations. Subclasses implement these methods according to their unique rules. The main class demonstrates deposits, withdrawals, overdrafts, credit usage, and balance display using **polymorphism**.

Background Theory

- **Abstract Class**: BankAccount serves as a template with common fields (accountNumber, balance) and an abstract method accountType().
- **Interface**: Transaction enforces the behavior of deposit() and withdraw() across all account types.
- **Inheritance**: SavingsAccount, CheckingAccount, and CreditAccount extend BankAccount, inheriting account attributes.
- **Polymorphism**: Each account is referenced as a Transaction type but executes its own overridden methods at runtime.
- **Method Overriding**: Different account types override deposit() and withdraw() to enforce specific rules.
- **Downcasting**: (BankAccount)acc1 is used to access the display() method since objects are referenced as Transaction.

Algorithm Design

1. **Define Abstract Class – BankAccount** ○ Attributes: accountNumber, balance.
   - Methods: display() for details, abstract method accountType().
2. **Define Interface – Transaction** ○ Methods: deposit(double amount), withdraw(double amount).

3. **Define Subclass – SavingsAccount** ○ Attribute: interestRate. ○ Deposit adds money, withdraw allowed only if balance ≥ amount. ○ Override accountType().
4. **Define Subclass – CheckingAccount** ○ Withdraw allows overdraft (balance can go negative).
    - Override accountType().
5. **Define Subclass – CreditAccount** ○ Attribute: creditLimit.
    - Withdraw allowed up to balance + creditLimit.
    - Override accountType().
6. **Main Class – BankDemo** ○ Create objects of SavingsAccount, CheckingAccount, and CreditAccount, referenced as Transaction. ○ Perform deposits and withdrawals. ○ Use downcasting to access display() for showing balances.

Conclusion

The Bank Account Management System demonstrates how **abstract classes and interfaces** can be combined to model real-world financial operations. Common attributes are centralized in BankAccount, while unique behaviors are implemented in subclasses. The Transaction interface ensures consistent deposit and withdrawal operations across all account types. Through **inheritance, polymorphism, and method overriding**, the program provides a flexible, reusable, and extensible design. It successfully models savings, checking, and credit accounts with their distinct rules, achieving the objectives of applying **OOP concepts** in a banking context.

Code & Output

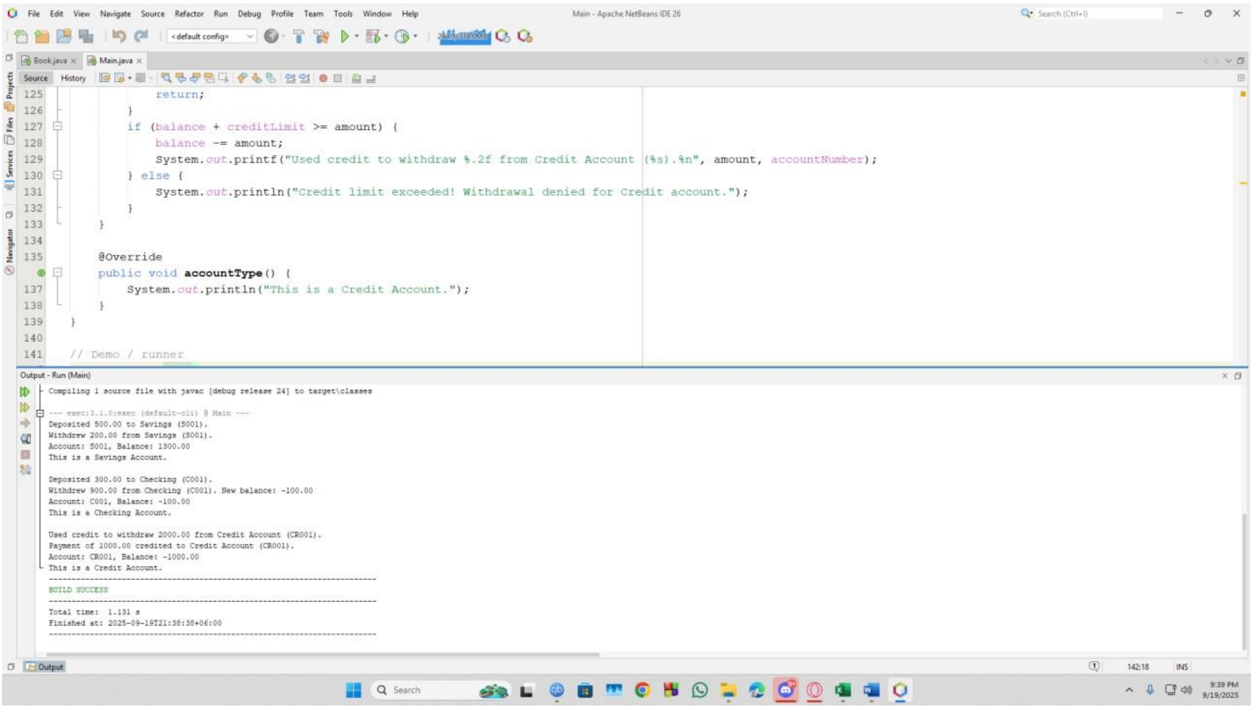
```
abstract class BankAccount {
protected String accountNumber;
   protected double balance;

   public BankAccount(String accountNumber, double initialBalance) {
this.accountNumber = accountNumber;
      this.balance = initialBalance;
}
   // Common display method (accessible after downcast when referenced by interface)
public void display() {
      System.out.printf("Account: %s, Balance: %.2f%n", accountNumber, balance);
   }
   // Each concrete account type describes itself
public abstract void accountType();
}
```

```java
interface Transaction {     void
deposit(double amount);     void
withdraw(double amount);
}

class SavingsAccount extends BankAccount implements Transaction {
private double interestRate = 0.03; // example field (not actively used here)

    public SavingsAccount(String accountNumber, double initialBalance) {
super(accountNumber, initialBalance);
    }
    @Override
    public void deposit(double amount) {
        if (amount <= 0) {
            System.out.println("Deposit amount must be positive.");
return;
        }
        balance += amount;
        System.out.printf("Deposited %.2f to Savings (%s).%n", amount, accountNumber);
    }
    @Override
    public void withdraw(double amount) {
        if (amount <= 0) {
            System.out.println("Withdraw amount must be positive.");
return;
        }
        if (balance >= amount) {
balance -= amount;
            System.out.printf("Withdrew %.2f from Savings (%s).%n", amount, accountNumber);
        } else {
            System.out.println("Insufficient balance! Withdrawal denied for Savings account.");
        }
}
    @Override
    public void accountType() {
        System.out.println("This is a Savings Account.");
    }
}
```

```
// Checking account: overdraft allowed (balance can go negative) class
CheckingAccount extends BankAccount implements Transaction {

    public CheckingAccount(String accountNumber, double initialBalance) {
super(accountNumber, initialBalance);
    }
    @Override
    public void deposit(double amount) {
        if (amount <= 0) {
            System.out.println("Deposit amount must be positive.");
            return;
        }
        balance += amount;
        System.out.printf("Deposited %.2f to Checking (%s).%n", amount, accountNumber);
    }
    @Override
    public void withdraw(double amount) {
        if (amount <= 0) {
            System.out.println("Withdraw amount must be positive.");
return;
        }
        balance -= amount; // overdraft allowed
        System.out.printf("Withdrew %.2f from Checking (%s). New balance: %.2f%n", amount,
accountNumber, balance);
    }
    @Override
    public void accountType() {
        System.out.println("This is a Checking Account.");
    }
}

// Credit account: has a credit limit
class CreditAccount extends BankAccount implements Transaction {
private double creditLimit = 5000.0;

    public CreditAccount(String accountNumber, double initialBalance) {
super(accountNumber, initialBalance);
    }
```

```
    public CreditAccount(String accountNumber, double initialBalance, double creditLimit) {
super(accountNumber, initialBalance);         this.creditLimit = creditLimit;
    }
    @Override
    public void deposit(double amount) {
        if (amount <= 0) {
            System.out.println("Deposit amount must be positive.");
return;
        }
        balance += amount;
        System.out.printf("Payment of %.2f credited to Credit Account (%s).%n", amount,
accountNumber);
    }
@O
verr
ide
    public void withdraw(double amount) {
        if (amount <= 0) {
            System.out.println("Withdraw amount must be positive.");
return;
        }
        if (balance + creditLimit >= amount) {
balance -= amount;
            System.out.printf("Used credit to withdraw %.2f from Credit Account (%s).%n", amount,
accountNumber);
        } else {
            System.out.println("Credit limit exceeded! Withdrawal denied for Credit account.");
        }
}
    @Override
    public void accountType() {
        System.out.println("This is a Credit Account.");
    }
}

// Demo / runner public
class BankDemo {
    public static void main(String[] args) {
        // Use the Transaction interface for polymorphism
        Transaction t1 = new SavingsAccount("S001", 1000.00);
```

```
        Transaction t2 = new CheckingAccount("C001", 500.00);
Transaction t3 = new CreditAccount("CR001", 0.00, 5000.00);

        // Savings operations
t1.deposit(500);        t1.withdraw(200);
        ((BankAccount) t1).display();
((BankAccount) t1).accountType();

        System.out.println();

        // Checking operations (overdraft allowed)
t2.deposit(300);        t2.withdraw(900); // allowed, balance
may go negative
        ((BankAccount) t2).display();
((BankAccount) t2).accountType();

        System.out.println();

        // Credit operations (uses credit limit)
        t3.withdraw(2000); // uses credit
t3.deposit(1000);           //      payment
((BankAccount) t3).display();
        ((BankAccount) t3).accountType();
    }
}
```



File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Main - Apache NetBeans IDE 26
Book.java
Main.java
Source
History
return;
}
if (balance + creditLimit >= amount) {
balance -= amount;
System.out.printf("Used credit to withdraw %.2f from Credit Account (%s).%n", amount, accountNumber);
} else {
System.out.println("Credit limit exceeded! Withdrawal denied for Credit account.");
}
}
@Override
public void accountType() {
System.out.println("This is a Credit Account.");
}
}
// Demo / runner
Output - Run (Main)
Compiling 1 source file with javac [debug release 24] to target\classes
--- exec:3.1.0:exec (default-cli) @ Main ---
Deposited 500.00 to Savings (S001).
Withdrew 200.00 from Savings (S001).
Account: S001, Balance: 1300.00
This is a Savings Account.
Deposited 300.00 to Checking (C001).
Withdrew 900.00 from Checking (C001). New balance: -100.00
Account: C001, Balance: -100.00
This is a Checking Account.
Used credit to withdraw 2000.00 from Credit Account (CR001).
Payment of 1000.00 credited to Credit Account (CR001).
Account: CR001, Balance: -1000.00
This is a Credit Account.
BUILD SUCCESS
Total time: 1.131 s
Finished at: 2025-09-19T21:38:38+06:00
Output
142:18
INS
Search
9:39 PM
9/19/2025