**BANK MANAGEMENT SYSTEM**

This Java program implements a simple banking system using object-oriented principles, threading, and synchronization. Here's a breakdown of how the code works:

**1. Interface TransactionOperations**

* It defines three operations that can be performed on an account: deposit(), withdraw(), and transfer().
* All accounts that implement this interface must define these methods.

**2. Abstract Class Account**

* Account is an abstract class that implements the TransactionOperations interface.
* It holds common attributes such as:
  + accountNumber: The account's unique identifier.
  + accountHolder: The name of the account holder.
  + balance: The account's current balance.
* The constructor initializes the account with an account number, holder's name, and initial deposit.
* **Methods**:
  + deposit(): Adds the specified amount to the balance. The method is synchronized to ensure thread safety.
  + withdraw(): Decreases the balance if there are sufficient funds. It is also synchronized.
  + transfer(): Transfers money from one account to another. It uses synchronization to avoid race conditions during the transfer.
  + displayAccountDetails(): An abstract method, forcing subclasses to provide account details.

**3. SavingsAccount Class**

* A subclass of Account, it adds an additional attribute:
  + interestRate: The interest rate for the savings account.
* Implements the displayAccountDetails() method, which prints all the account details, including the interest rate.

**4. CurrentAccount Class**

* Another subclass of Account, with an additional attribute:
  + overdraftLimit: Allows the account to withdraw more money than the balance, up to this limit.
* Overrides the withdraw() method to allow for overdrafts, checking if the total of the balance and overdraft limit is sufficient for the withdrawal.

**5. TransactionTask Class (Implements Runnable)**

* This class handles the execution of transactions in separate threads.
* It takes a TransactionOperations object, the type of transaction (deposit, withdraw, or transfer), the amount, and an optional toAccount for transfers.
* The run() method checks the type of transaction and executes the corresponding method.

**6. Main Class (BankManagementSystem)**

* In the main() method:
  1. Two accounts are created: a SavingsAccount for Alice and a CurrentAccount for Bob.
  2. Their details are displayed.
  3. Four threads are started, performing the following operations:
     + **Thread 1** deposits 500 into Alice's savings account.
     + **Thread 2** withdraws 1500 from Bob's current account.
     + **Thread 3** transfers 200 from Alice's savings to Bob's current account.
     + **Thread 4** deposits 300 into Bob's current account.
  4. The threads are joined to ensure that the main thread waits for them to complete before proceeding.
  5. After all transactions are completed, the final account details are displayed.

**Multithreading and Synchronization**

* The synchronized keyword is used in deposit, withdraw, and transfer operations to prevent race conditions when multiple threads try to access and modify shared resources (account balance).
* Each transaction runs in a separate thread, simulating real-world scenarios where multiple users could be performing operations on the same account at the same time.

**Creating the Project Report**

For your project report, you can use the following structure:

1. **Introduction**:
   * Explain the purpose of the banking system project, highlighting how it simulates real-world banking operations with multithreading.
2. **Overview of Existing Solutions**:
   * Discuss how real-world banking applications handle multiple transactions concurrently, ensuring data integrity.
3. **Limitations of Existing System**:
   * Mention issues with race conditions in multi-threaded environments and the need for synchronization.
4. **Need for New System**:
   * Justify the use of a synchronized, thread-safe system to avoid problems such as inconsistent balance updates.
5. **System Design**:
   * Provide a UML diagram of your class structure, showing TransactionOperations, Account, SavingsAccount, CurrentAccount, and TransactionTask.
   * Explain how threads are created and managed using the Runnable interface.
6. **Implementation**:
   * Describe the code in detail, as explained above.
   * Highlight how synchronization ensures thread safety.
7. **Results & Discussions**:
   * Show sample outputs of the program, demonstrating successful deposits, withdrawals, and transfers.
   * Discuss how the system behaves under concurrent transactions.
8. **Conclusion**:
   * Summarize the key takeaways, emphasizing the importance of thread safety in financial systems.

**CODE:**

interface TransactionOperations {

void deposit(double amount);

void withdraw(double amount);

void transfer(Account toAccount, double amount);

}

abstract class Account implements TransactionOperations {

protected String accountNumber;

protected String accountHolder;

protected double balance;

public Account(String accountNumber, String accountHolder, double initialDeposit) {

this.accountNumber = accountNumber;

this.accountHolder = accountHolder;

this.balance = initialDeposit;

}

public synchronized void deposit(double amount) {

balance += amount;

System.out.println(Thread.currentThread().getName() + " deposited " + amount + ". New Balance: " + balance);

}

public synchronized void withdraw(double amount) {

if (balance >= amount) {

balance -= amount;

System.out.println(Thread.currentThread().getName() + " withdrew " + amount + ". New Balance: " + balance);

} else {

System.out.println("Insufficient funds for " + Thread.currentThread().getName());

}

}

public void transfer(Account toAccount, double amount) {

synchronized (this) {

if (balance >= amount) {

this.withdraw(amount);

toAccount.deposit(amount);

System.out.println(Thread.currentThread().getName() + " transferred " + amount + " to " + toAccount.accountNumber);

} else {

System.out.println("Transfer failed due to insufficient funds for " + Thread.currentThread().getName());

}

}

}

public abstract void displayAccountDetails();

}

class SavingsAccount extends Account {

private double interestRate;

public SavingsAccount(String accountNumber, String accountHolder, double initialDeposit, double interestRate) {

super(accountNumber, accountHolder, initialDeposit);

this.interestRate = interestRate;

}

@Override

public void displayAccountDetails() {

System.out.println("Savings Account [Account Number: " + accountNumber + ", Holder: " + accountHolder +

", Balance: " + balance + ", Interest Rate: " + interestRate + "%]");

}

}

class CurrentAccount extends Account {

private double overdraftLimit;

public CurrentAccount(String accountNumber, String accountHolder, double initialDeposit, double overdraftLimit) {

super(accountNumber, accountHolder, initialDeposit);

this.overdraftLimit = overdraftLimit;

}

@Override

public synchronized void withdraw(double amount) {

if (balance + overdraftLimit >= amount) {

balance -= amount;

System.out.println(Thread.currentThread().getName() + " withdrew " + amount + ". New Balance: " + balance);

} else {

System.out.println("Overdraft limit exceeded for " + Thread.currentThread().getName());

}

}

@Override

public void displayAccountDetails() {

System.out.println("Current Account [Account Number: " + accountNumber + ", Holder: " + accountHolder +

", Balance: " + balance + ", Overdraft Limit: " + overdraftLimit + "]");

}

}

class TransactionTask implements Runnable {

private TransactionOperations operation;

private String type;

private double amount;

private Account toAccount;

public TransactionTask(TransactionOperations operation, String type, double amount) {

this.operation = operation;

this.type = type;

this.amount = amount;

}

public TransactionTask(TransactionOperations operation, String type, double amount, Account toAccount) {

this(operation, type, amount);

this.toAccount = toAccount;

}

@Override

public void run() {

switch (type.toLowerCase()) {

case "deposit":

operation.deposit(amount);

break;

case "withdraw":

operation.withdraw(amount);

break;

case "transfer":

if (operation instanceof Account && toAccount != null) {

((Account) operation).transfer(toAccount, amount);

}

break;

default:

System.out.println("Invalid transaction type");

}

}

}

public class BankManagementSystem {

public static void main(String[] args) {

SavingsAccount savings = new SavingsAccount("SA123", "Alice", 1000.0, 5.0);

CurrentAccount current = new CurrentAccount("CA456", "Bob", 2000.0, 500.0);

savings.displayAccountDetails();

current.displayAccountDetails();

Thread t1 = new Thread(new TransactionTask(savings, "deposit", 500.0), "Thread-1");

Thread t2 = new Thread(new TransactionTask(current, "withdraw", 1500.0), "Thread-2");

Thread t3 = new Thread(new TransactionTask(savings, "transfer", 200.0, current), "Thread-3");

Thread t4 = new Thread(new TransactionTask(current, "deposit", 300.0), "Thread-4");

t1.start();

t2.start();

t3.start();

t4.start();

try {

t1.join();

t2.join();

t3.join();

t4.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("\nFinal Account Details:");

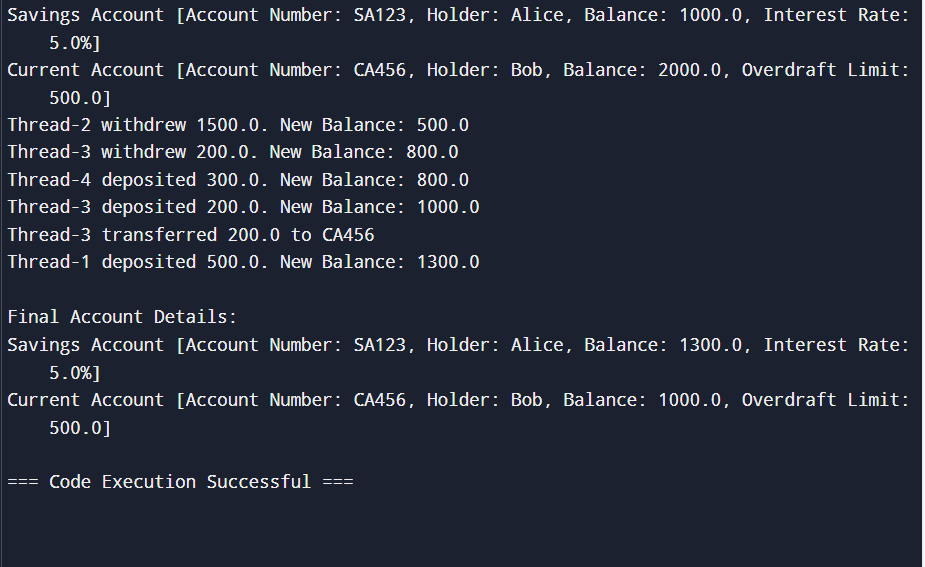
savings.displayAccountDetails();

current.displayAccountDetails();

}

}

**OUTPUT:**

****