**CASE STUDY**

**BANKING SYSTEM**

Submitted by:

E.J. G. DOMINGO

J. T. OCAMPO

*Bachelor of Science in Computer Science Student*

BSCS2C

Submitted to:  
  
**Mr. Matt Wage Flordeliza***Instructor*

May 17, 2024

**Introduction**

This study details the creation of a Java-based banking system. Customers can control their accounts, deposit or withdraw funds, transfer money, and close accounts using the system. There are two kinds of accounts available: Savings and Checking. Savings accounts enable individuals to earn interest on their funds, while Checking accounts provide overdraft protection, permitting clients to withdraw money exceeding their balance to a certain extent. The goal of the project is to replicate fundamental banking activities in a straightforward and effective manner.

The framework is structured based on fundamental Object-Oriented Programming (OOP) concepts like inheritance, encapsulation, and code reuse. The SavingsAccount and CheckingAccount classes both derive common attributes from the Account class, making the code simpler and allowing for easy addition of new features. Encapsulation guarantees that confidential information, such as account balances, remains confidential and can only be altered or viewed using secure procedures like deposit and withdraw. This guarantees that the system manages transactions accurately and securely.

Clients have the ability to create and oversee numerous accounts in the platform, reflecting common banking behaviors where individuals frequently have both a Savings and Checking account. The system guarantees that transactions are conducted with security, avoiding mistakes such as overdrawing without overdraft protection or trying to move money between accounts that are not valid. The system replicates the advantages and drawbacks of handling Savings and Checking accounts by incorporating functions such as interest computation and overdraft protection.

This project shows the utilization of Java in creating a practical banking system that mirrors real-life banking operations. Inheritance and encapsulation help maintain organization and security in the system, while code reuse promotes efficient development and simple maintenance. Using object-oriented programming concepts to simulate core banking functions demonstrates how scalable and practical solutions can be created for real-world applications.

**Case Analysis**

The creation of this banking system aimed to imitate regular banking activities such as account management, money deposits and withdrawals, and transfers, through the application of fundamental Object-Oriented Programming (OOP) principles. The system is designed to have two types of accounts—Savings and Checking—both of which inherit common functions such as deposits and withdrawals. A key advantage of this framework is its capacity to decrease redundant code and simplify system upkeep. The importance of encapsulation lies in its protection of sensitive data, like account balances, to guarantee security and restrict access to specific methods, thus ensuring system functionality and reducing errors. Furthermore, the distinct categorization of account varieties allows for the incorporation of specific characteristics designed for each account, improving the overall effectiveness. By adhering to these OOP principles, the system not only carries out its intended functions but also establishes a strong foundation for future improvements. In general, carefully applying OOP concepts greatly enhances the system's dependability and user-friendliness.

There are certain restrictions in the existing execution. Customers can only have up to 10 accounts, which may not be sufficient in situations where people require greater flexibility. While the system currently manages overdrafts and interest on Savings accounts well, enhancing user input validation and error-handling features would enhance its user-friendliness. Additionally, enhancing the system to include other financial services such as loan administration or offering a comprehensive transaction history may strengthen the system and better reflect actual banking practices.

**Documentation**

The process of documenting the banking system started with identifying the necessary components to include. Precise and comprehensive explanations were provided for every class and method to help users comprehend the code and guarantee that anyone analyzing it could understand its layout and features. Examples were provided to demonstrate how customers can open accounts and conduct transactions. In general, documenting the project improved clarity on objectives and emphasized the system's functionality, facilitating presentation in the case study.

Furthermore, maintaining clear and organized documentation was crucial in order to ensure readability. Concise titles and uncomplicated paragraphs made complex concepts easier to comprehend. A list was used to monitor what tasks had been finished and what tasks were still pending. During the recording of the testing stages, the team made a record of typical user interactions, anticipated outcomes, and potential error notifications. Emphasizing actual situations demonstrated the push to create a more user-friendly banking system.

Efficient documentation was determined to be crucial for the ongoing success of the banking industry. This proactive method is a helpful tool for those wanting to upgrade or improve the system in the future. By intensely documenting the process, the aim was to establish a project that was operational, clear, and provided a strong base for future advancements.

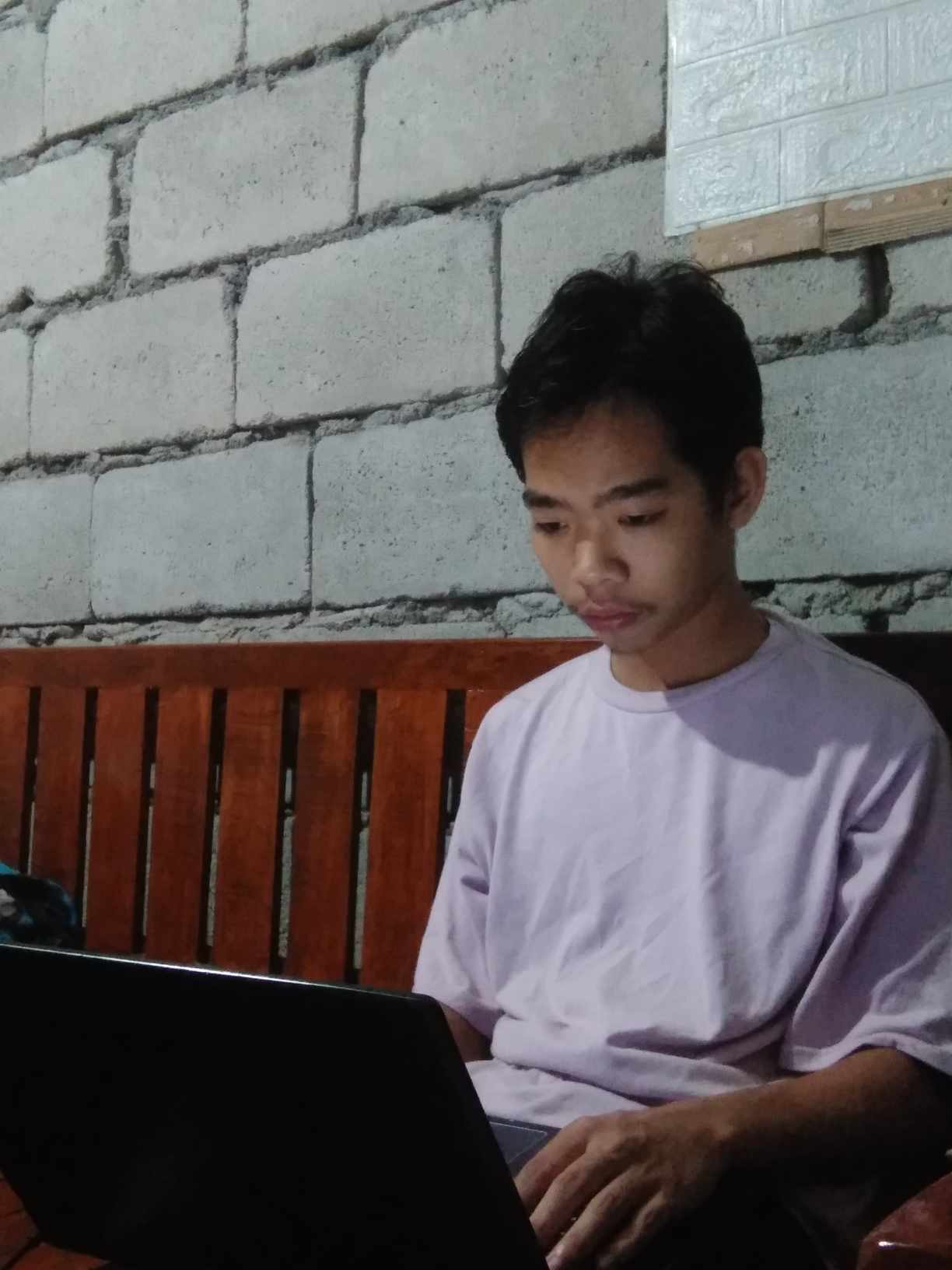
**Brainstorming**

During the brainstorming phrase, researchers identified key aspects of the bank system, including customer account management and information, transaction tracking and record-keeping, account offerings and services, and employee data management. The bank system aims to provide user-friendly features, scalability and high availability for both customers and the bank organization.

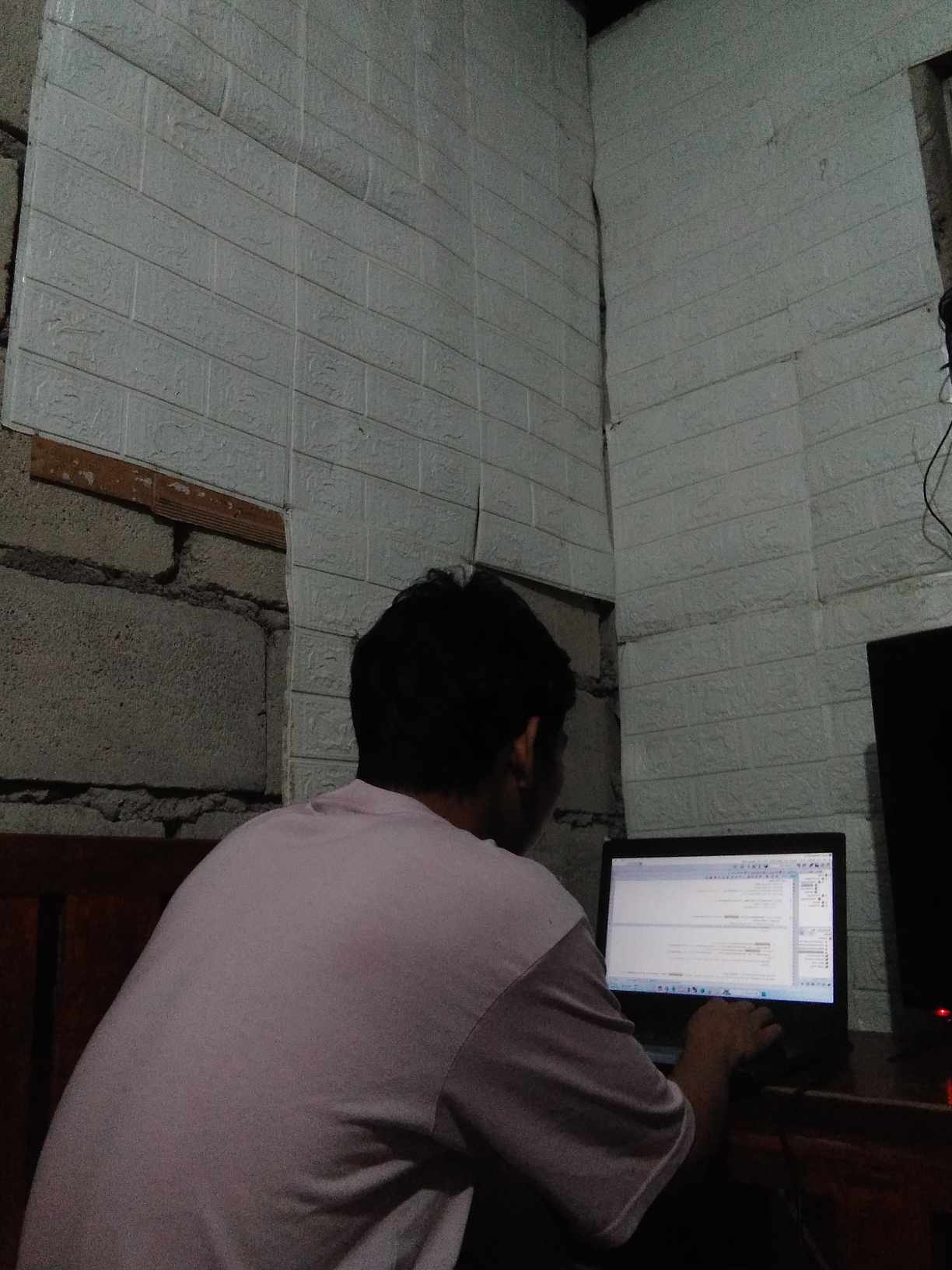
Possible challenges were recognized, such us ensuring data security and confidentiality and achieving customer satisfaction and trust. To address these issue researchers proposed solutions, including implementing a data description and backup approach, designing a system that can adapt to changing customer needs and growth and conducting multiple tests to ensure system reliability and security.

Regular online meetings were held to discuss solutions to system issues and bugs, customize the system meets the customer needs, and monitor system functionality, efficiency and customer satisfaction. The brainstorming phase highlighted the essential factors required for the bank system`success.

**Pictures**



Picture 1: Planning how the code looks like based on the instructions given.



Picture 2: Start implementing the code in netbeans.



Picture 3: The case study is starting to be worked on.



Picture 4: :Checking both the case study and the source code.

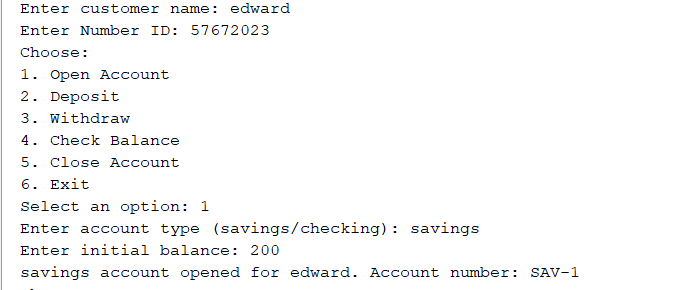


Picture 5: Finalizing both the case study and the source code.

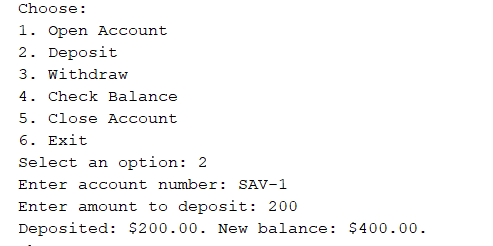
**Solution**

In response to the challenges identified in the case analysis, a comprehensive solution has been proposed involving the development of a Java-based banking system. This solution automates account management, allowing customers to create, update, and delete accounts seamlessly. Additionally, it incorporates online banking features, enabling customers to perform deposits and withdrawals through a command-line interface. By implementing this banking system, ABC Bank can significantly enhance its operational efficiency, reduce transaction processing times, and improve customer satisfaction. The foundational structure of the system not only addresses current needs but also positions the bank for future growth by allowing for the integration of additional features such as transaction history tracking and advanced online banking services. This strategic approach aims to foster customer loyalty and ensure the bank remains competitive in the evolving financial landscape.

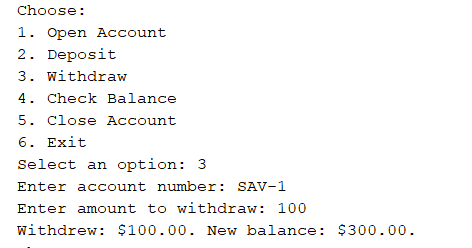
**Screenshot of Input and Output**

****

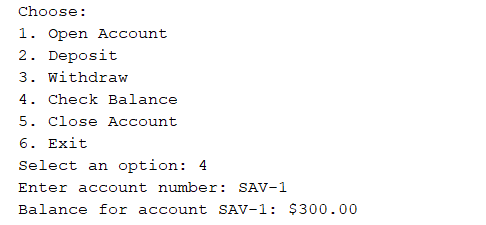
Screenshot 1: Illustrates an automated system where Edward efficiently opens a savings account, showcasing how digital banking simplifies processes and enhances customer experience.

****

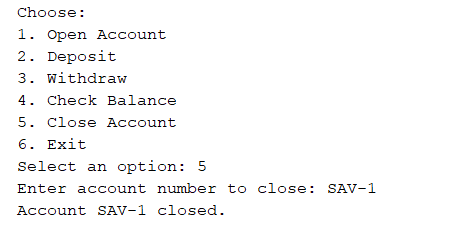
Screenshot 2: A customer using an automated banking system to deposit $200 into account number "SAV-1," resulting in a new balance of $400. This highlights the ease and efficiency of using digital banking to perform transactions quickly.

****

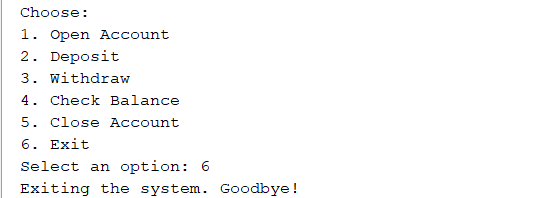
Screenshot 3: A customer using an automated banking system to withdraw $100 into account number "SAV-1," resulting in a new balance of $300. This highlights the ease and efficiency of using digital banking to perform transactions quickly.

****

Screenshot 4: A customer using an automated banking system to check the balance of account number "SAV-1," resulting the balance of account number “SAV-1” is $300.

****

Screenshot 5: A customer using an automated banking system to close the account number "SAV-1,".

****

Screenshot 5: A customer using an automated banking system to exit the program.

**Conclusion**

The banking system is a critical segment of developed economies, laying a basis for financial stability and growth. Simultaneously, through the management of deposits, giving credits, and making transactions safer, banks play both roles at once-for personal well-being and business development. Ease of access and convenience in digital banking continues to be improved upon, with services now being available to the global community with just one click. Today, the banking system, therefore, serves as the platform on which efficient management of risks and resources helps pursue economic stability and progress around the world.

**Reference**

Deitel, P. J. & Deitel, H. M. (2016) *Java: How to Program*. 11th ed. Upper Saddle River, NJ: Pearson Education. Available at: [https://deitel.com/java-how-to-program-11-e-early-objects-version/](https://deitel.com/java-how-to-program-11-e-early-objects-version/%20) (Accessed: 10 October 2024).

Lee, J. (2019) Object-Oriented Programming in Java: An Introduction. New York, NY: Wiley. Available at: [https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java/](https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java/%20) (Accessed: 10 October 2024).  
Oracle (n.d.) Java Documentation. Available at: <https://docs.oracle.com/en/java/> (Accessed: 12 October 2024).  
GeeksforGeeks (2024) Object-Oriented Programming (OOP) Concept in Java. Available at: [https://www.geeksforgeeks.org/object-oriented-programming-oops-java/](https://www.geeksforgeeks.org/object-oriented-programming-oops-java/%20) (Accessed: 12 October 2024).

**Source Code**

**MAIN CLASS**

import java.util.Scanner;

public class Main {

private static Customer customer;

public static void main(String[] args) {

Scanner scan = new Scanner(System.in);

System.out.print("Enter customer name: ");

String name = scan.nextLine();

System.out.print("Enter Number ID: ");

String numId = scan.nextLine();

customer = new Customer(name, numId);

String option;

do {

System.out.println("Choose:");

System.out.println("1. Open Account");

System.out.println("2. Deposit");

System.out.println("3. Withdraw");

System.out.println("4. Check Balance");

System.out.println("5. Close Account");

System.out.println("6. Exit");

System.out.print("Select an option: ");

option = scan.nextLine();

switch (option) {

case "1":

openAccount(scan);

break;

case "2":

deposit(scan);

break;

case "3":

withdraw(scan);

break;

case "4":

checkBalance(scan);

break;

case "5":

closeAccount(scan);

break;

case "6":

System.out.println("Exiting the system. Goodbye!");

System.exit(0);

break;

default:

System.out.println("Invalid option. Please try again.");

}

} while (!option.equals("6"));

scan.close();

}

private static void openAccount(Scanner scanner) {

System.out.print("Enter account type (savings/checking): ");

String accountType = scanner.nextLine();

System.out.print("Enter initial balance: ");

double initialBalance = scanner.nextDouble();

scanner.nextLine(); // Consume newline

customer.openAccount(accountType, initialBalance);

}

private static void deposit(Scanner scanner) {

System.out.print("Enter account number: ");

String accountNumber = scanner.nextLine();

Account account = customer.findAccount(accountNumber);

if (account != null) {

System.out.print("Enter amount to deposit: ");

double amount = scanner.nextDouble();

scanner.nextLine(); // Consume newline

account.deposit(amount);

} else {

System.out.println("Account not found.");

}

}

private static void withdraw(Scanner scanner) {

System.out.print("Enter account number: ");

String accountNumber = scanner.nextLine();

Account account = customer.findAccount(accountNumber);

if (account != null) {

System.out.print("Enter amount to withdraw: ");

double amount = scanner.nextDouble();

scanner.nextLine(); // Consume newline

if (account instanceof CheckingAccount) {

((CheckingAccount) account).overdraftProtection(amount);

} else {

account.withdraw(amount);

}

} else {

System.out.println("Account not found.");

}

}

private static void checkBalance(Scanner scanner) {

System.out.print("Enter account number: ");

String accountNumber = scanner.nextLine();

Account account = customer.findAccount(accountNumber);

if (account != null) {

System.out.printf("Balance for account %s: $%.2f%n", account.getAccountNumber(), account.getBalance());

} else {

System.out.println("Account not found.");

}

}

private static void closeAccount(Scanner scanner) {

System.out.print("Enter account number to close: ");

String accountNumber = scanner.nextLine();

customer.closeAccount(accountNumber);

}

}

**SUB CLASS**

class Account {

private String accountNumber;

private double balance;

public Account(String accountNumber, double initialBalance) {

this.accountNumber = accountNumber;

this.balance = initialBalance;

}

public double getBalance() {

return balance;

}

public String getAccountNumber() {

return accountNumber;

}

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

System.out.printf("Deposited: $%.2f. New balance: $%.2f.%n", amount, balance);

}

}

public void withdraw(double amount) {

if (amount > 0) {

if (amount <= balance) {

balance -= amount;

System.out.printf("Withdrew: $%.2f. New balance: $%.2f.%n", amount, balance);

} else {

System.out.println("Insufficient funds.");

}

} else {

System.out.println("Withdrawal amount must be positive.");

}

}

}

class SavingsAccount extends Account {

public SavingsAccount(String accountNumber, double initialBalance) {

super(accountNumber, initialBalance);

}

public void addInterest(double interestRate) {

double interest = getBalance() \* interestRate;

deposit(interest);

System.out.printf("Interest of $%.2f added to account %s.%n", interest, getAccountNumber());

}

}

class CheckingAccount extends Account {

public CheckingAccount(String accountNumber, double initialBalance) {

super(accountNumber, initialBalance);

}

public void overdraftProtection(double amount) {

if (amount > 0) {

if (amount <= getBalance()) {

withdraw(amount);

} else {

System.out.printf("Overdraft protection activated. Cannot withdraw $%.2f due to insufficient funds.%n", amount);

}

} else {

System.out.println("Withdrawal amount must be positive.");

}

}

}

class Customer {

private String name;

private String numId;

private Account[] accounts = new Account[5]; // Up to 5 accounts

private int accountCount = 0;

public Customer(String name, String customerId) {

this.name = name;

this.numId = customerId;

}

public void openAccount(String accountType, double initialBalance) {

Account newAccount;

if (accountCount >= accounts.length) {

System.out.println("Maximum account limit reached.");

return;

}

if (accountType.equalsIgnoreCase("savings")) {

newAccount = new SavingsAccount("SAV-" + (accountCount + 1), initialBalance);

} else if (accountType.equalsIgnoreCase("checking")) {

newAccount = new CheckingAccount("CHK-" + (accountCount + 1), initialBalance);

} else {

System.out.println("Invalid account type. Choose 'savings' or 'checking'.");

return;

}

accounts[accountCount++] = newAccount;

System.out.printf("%s account opened for %s. Account number: %s%n", accountType, name, newAccount.getAccountNumber());

}

public void closeAccount(String accountNumber) {

for (int i = 0; i < accountCount; i++) {

if (accounts[i].getAccountNumber().equals(accountNumber)) {

accounts[i] = accounts[--accountCount]; // Remove the account

accounts[accountCount] = null; // Avoid duplication

System.out.printf("Account %s closed.%n", accountNumber);

return;

}

}

System.out.println("Account not found.");

}

public Account findAccount(String accountNumber) {

for (int i = 0; i < accountCount; i++) {

if (accounts[i].getAccountNumber().equals(accountNumber)) {

return accounts[i];

}

}

return null;

}

}