PROJECT

Student Name: Arti | **UID: 22BCA10812**
Branch: BCA | **Section/Group: 3-B**
Semester: 5th | **Date of Performance: 24/10/2024**
Subject Name: Web Development | **Subject Code: 22CAH-301**

1. **Aim/Overview of the project.**

   The Bank Management System is designed to facilitate the management of accounts in a bank. This application provides functionalities for both administrators and customers, streamlining the processes of adding accounts, searching for accounts, depositing, withdrawing, and viewing account balances.

2. **Task to be done.**

   (a) Account Management:
   • Allow administrators to add new accounts to the bank database, including details such as account number, account holder's name, and initial balance.
   • Display all accounts in the bank along with their current status (active or inactive).

   (b) Search and Query:
   • Enable users to search for specific accounts by account number.
   • Provide functionality to list all accounts, making it easier for users to view account details.

   (c) Depositing and Withdrawing Money:
   • Allow customers to deposit money into their accounts, updating the balance accordingly.
   • Facilitate the withdrawal of money from customer accounts, ensuring sufficient balance for the transaction.

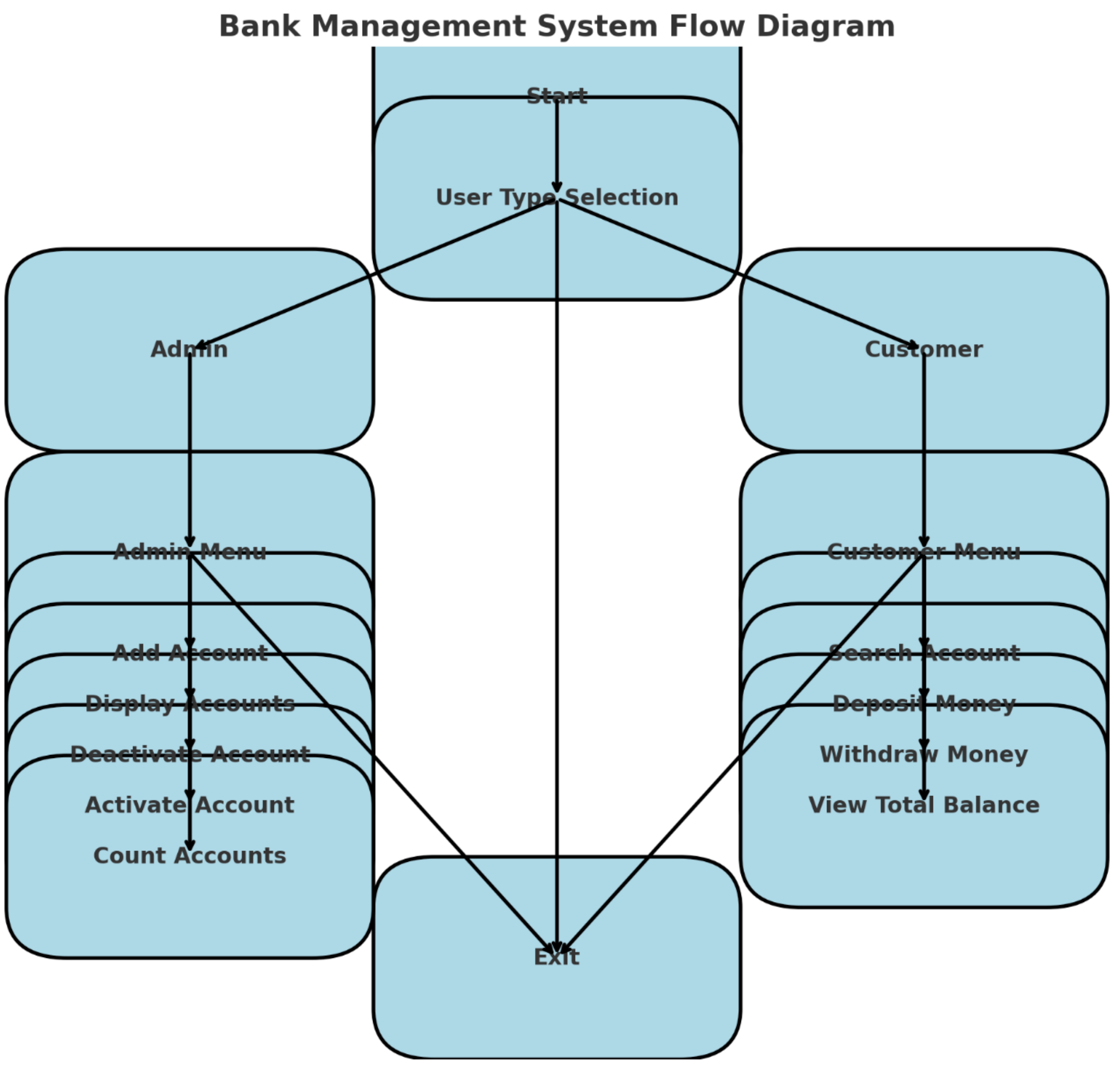
(d) User-Friendly Interface:

• Implement a clear and intuitive menu system for both admin and customer users to navigate through the application's features.

(e) Scalability:

• Design the system to handle a limited number of accounts (up to 100), with the potential to scale up by adjusting the underlying data structure.

3. Algorithm/Flowchart.



4. Dataset.

a) Account Data Set

Each account in the bank is represented by an instance of the `Account` class, which holds the following data:

Account Number: A string representing the unique account identifier.

Account Holder Name: A string indicating the name of the account holder.

Balance: A float representing the current balance in the account.

IsActive: A boolean indicating whether the account is active or inactive**.**

b) Bank Data Set

The `Bank` class contains a collection of `Account` objects stored in a list. This list serves as the primary dataset for managing the bank's account details:

Array/List of Accounts: A dynamic list (`List<Account> accounts`) that can hold multiple `Account` instances (up to a predefined maximum).

Count of Accounts: An integer representing the total number of accounts currently in the bank.

c) User Interactions

While not part of the dataset itself, user interactions with the system contribute to the state of the data. This includes actions like:

Adding new accounts to the dataset.

Searching for accounts within the dataset.

Depositing or withdrawing money, which updates the `balance` in the dataset.

Activating or deactivating accounts, which updates the `IsActive` status in the dataset.

5. **Code for experiment/practical:**

```
  using System;
using System.Collections.Generic;

class Account
{
    public string AccountNumber { get; set; }
    public string AccountHolderName { get; set; }
    public float Balance { get; set; }
    public bool IsActive { get; set; }

    public Account(string accNum = "", string accHolder = "", float bal = 0.0f)
    {
        AccountNumber = accNum;
        AccountHolderName = accHolder;
        Balance = bal;
        IsActive = true; // Initialize isActive to true
    }
}

class Bank
{
    private const int MAX_ACCOUNTS = 100; // Maximum number of accounts
private List<Account> accounts; // List to store Account objects
private int count; // Current number of accounts


public Bank()
{
   accounts = new List<Account>(); // Initialize the list
   count = 0; // Initialize count to 0
}


public void AddAccount(Account account)
{
   if (count < MAX_ACCOUNTS)
   {
      accounts.Add(account); // Add the account to the list
      count++;
      Console.WriteLine("Account added successfully!");
   }
   else
   {
      Console.WriteLine("Bank is full, cannot add more accounts.");
   }
}


public void DisplayAccounts()
```

```
    {
        if (count == 0)
        {
            Console.WriteLine("No accounts in the bank.");
            return;
        }

        Console.WriteLine("Accounts in the Bank:");
        foreach (var account in accounts)
        {
            Console.WriteLine($"Account Number: {account.AccountNumber}, Account
Holder: {account.AccountHolderName}, Balance: ${account.Balance}, Active:
{(account.IsActive ? "Yes" : "No")}");
        }
    }

    public void SearchAccount(string accountNumber)
    {
        foreach (var account in accounts)
        {
            if (account.AccountNumber == accountNumber)
            {
                Console.WriteLine("Account found:");
                Console.WriteLine($"Account Number: {account.AccountNumber}, Account
Holder: {account.AccountHolderName}, Balance: ${account.Balance}, Active:
{(account.IsActive ? "Yes" : "No")}");

                return;

            }

        }

        Console.WriteLine("Account not found.");

    }


    public void ViewTotalBalance(string accountNumber)

    {

        foreach (var account in accounts)

        {

            if (account.AccountNumber == accountNumber)

            {

                Console.WriteLine($"Total Balance for account {accountNumber} is:
${account.Balance}");

                return;

            }

        }

        Console.WriteLine("Account not found.");

    }


    public void DepositMoney(string accountNumber, float amount)

    {
        foreach (var account in accounts)
        {
            if (account.AccountNumber == accountNumber)
            {
                if (account.IsActive)
                {
                    account.Balance += amount; // Add the deposit amount to balance
                    Console.WriteLine($"Deposited ${amount} into account {accountNumber}");
                    Console.WriteLine($"New balance: ${account.Balance}");
                }
                else
                {
                    Console.WriteLine($"Account {accountNumber} is inactive.");
                }
                return;
            }
        }
        Console.WriteLine("Account not found.");
    }

    public void WithdrawMoney(string accountNumber, float amount)
    {
        foreach (var account in accounts)
        {
            if (account.AccountNumber == accountNumber)
            {
                if (account.IsActive)
                {
                    if (account.Balance >= amount)
                    {
                        account.Balance -= amount; // Subtract the withdrawal amount from
balance
                        Console.WriteLine($"Withdrew ${amount} from account
{accountNumber}");
                        Console.WriteLine($"New balance: ${account.Balance}");
                    }
                    else
                    {
                        Console.WriteLine($"Insufficient balance in account {accountNumber}");
                    }
                }
                else
                {
                    Console.WriteLine($"Account {accountNumber} is inactive.");
                }
                return;
            }
        }
        Console.WriteLine("Account not found.");
```

```
}


public void DeactivateAccount(string accountNumber)
{
    foreach (var account in accounts)
    {
        if (account.AccountNumber == accountNumber)
        {
            if (account.IsActive)
            {
                account.IsActive = false; // Mark account as inactive
                Console.WriteLine($"Account {accountNumber} has been deactivated.");
            }
            else
            {
                Console.WriteLine($"Account {accountNumber} is already inactive.");
            }
            return;
        }
    }
    Console.WriteLine("Account not found.");
}


public void ActivateAccount(string accountNumber)
```

```
{
    foreach (var account in accounts)
    {
        if (account.AccountNumber == accountNumber)
        {
            if (!account.IsActive)
            {
                account.IsActive = true; // Mark account as active
                Console.WriteLine($"Account {accountNumber} has been activated.");
            }
            else
            {
                Console.WriteLine($"Account {accountNumber} is already active.");
            }
            return;
        }
    }
    Console.WriteLine("Account not found.");
}

public int GetAccountCount()
{
    return count;
}
```

```
}


class Program
{
    static void AdminMenu(Bank bank)
    {
        int choice;
        do
        {
            Console.WriteLine("\nAdmin Menu");
            Console.WriteLine("1. Add Account");
            Console.WriteLine("2. Display Accounts");
            Console.WriteLine("3. Deactivate Account");
            Console.WriteLine("4. Activate Account");
            Console.WriteLine("5. Count Accounts in Bank");
            Console.WriteLine("6. Exit");
            Console.Write("Enter your choice: ");
            choice = int.Parse(Console.ReadLine());

            switch (choice)
            {
                case 1:
                    Console.Write("Enter account number: ");
                    string accountNumber = Console.ReadLine();
    Console.Write("Enter account holder name: ");
    string accountHolder = Console.ReadLine();
    Console.Write("Enter initial balance: ");
    float balance = float.Parse(Console.ReadLine());
    bank.AddAccount(new Account(accountNumber, accountHolder, balance));
    break;

case 2:
    bank.DisplayAccounts();
    break;

case 3:
    Console.Write("Enter account number to deactivate: ");
    accountNumber = Console.ReadLine();
    bank.DeactivateAccount(accountNumber);
    break;

case 4:
    Console.Write("Enter account number to activate: ");
    accountNumber = Console.ReadLine();
    bank.ActivateAccount(accountNumber);
    break;

case 5:
```

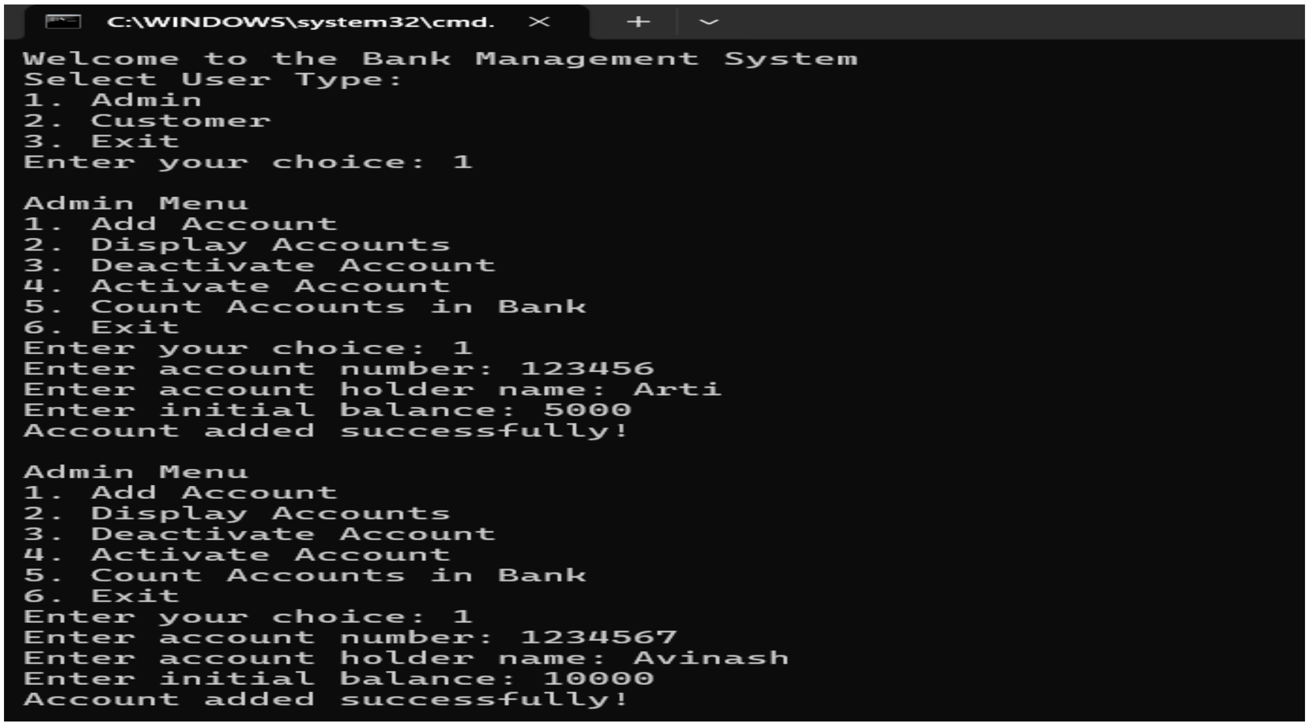
```
                Console.WriteLine($"Total accounts in bank: {bank.GetAccountCount()}");
                break;

            case 6:
                Console.WriteLine("Exiting admin menu...");
                break;

            default:
                Console.WriteLine("Invalid choice! Please try again.");
                break;
        }
    } while (choice != 6);
}

static void CustomerMenu(Bank bank)
{
    int choice;
    do
    {
        Console.WriteLine("\nCustomer Menu");
        Console.WriteLine("1. Search Account");
        Console.WriteLine("2. Deposit Money");
        Console.WriteLine("3. Withdraw Money");
        Console.WriteLine("4. View Total Balance");
Console.WriteLine("5. Exit");
Console.Write("Enter your choice: ");
choice = int.Parse(Console.ReadLine());

switch (choice)
{
    case 1:
        Console.Write("Enter account number to search: ");
        string accountNumber = Console.ReadLine();
        bank.SearchAccount(accountNumber);
        break;

    case 2:
        Console.Write("Enter account number to deposit: ");
        accountNumber = Console.ReadLine();
        Console.Write("Enter amount to deposit: ");
        float amount = float.Parse(Console.ReadLine());
        bank.DepositMoney(accountNumber, amount);
        break;

    case 3:
        Console.Write("Enter account number to withdraw from: ");
        accountNumber = Console.ReadLine();
        Console.Write("Enter amount to withdraw: ");
```

```
                amount = float.Parse(Console.ReadLine());
                bank.WithdrawMoney(accountNumber, amount);
                break;

            case 4:
                Console.Write("Enter account number to view balance: ");
                accountNumber = Console.ReadLine();
                bank.ViewTotalBalance(accountNumber);
                break;

            case 5:
                Console.WriteLine("Exiting customer menu...");
                break;

            default:
                Console.WriteLine("Invalid choice! Please try again.");
                break;
        }
    } while (choice != 5);
}

static void Main(string[] args)
{
    Bank bank = new Bank();
```

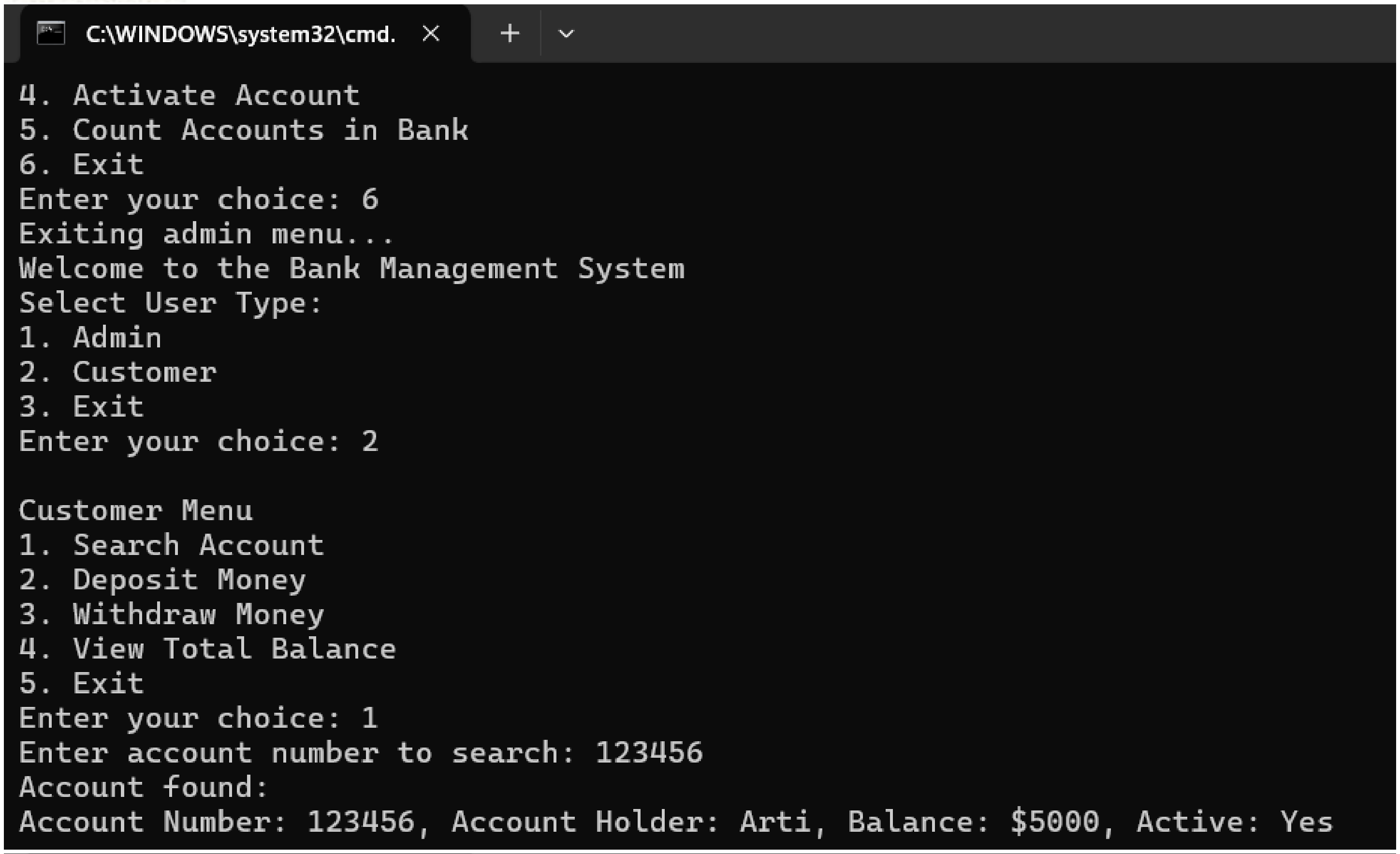
```
int userType;


do
{
   Console.WriteLine("Welcome to the Bank Management System");
   Console.WriteLine("Select User Type: ");
   Console.WriteLine("1. Admin");
   Console.WriteLine("2. Customer");
   Console.WriteLine("3. Exit");
   Console.Write("Enter your choice: ");
   userType = int.Parse(Console.ReadLine());


   switch (userType)
   {
      case 1:
         AdminMenu(bank);
         break;


      case 2:
         CustomerMenu(bank);
         break;


      case 3:
         Console.WriteLine("Exiting the system...");
```

```
                break;


            default:

                Console.WriteLine("Invalid user type selected. Please try again.");

                break;

        }

    } while (userType != 3);

  }

}
```

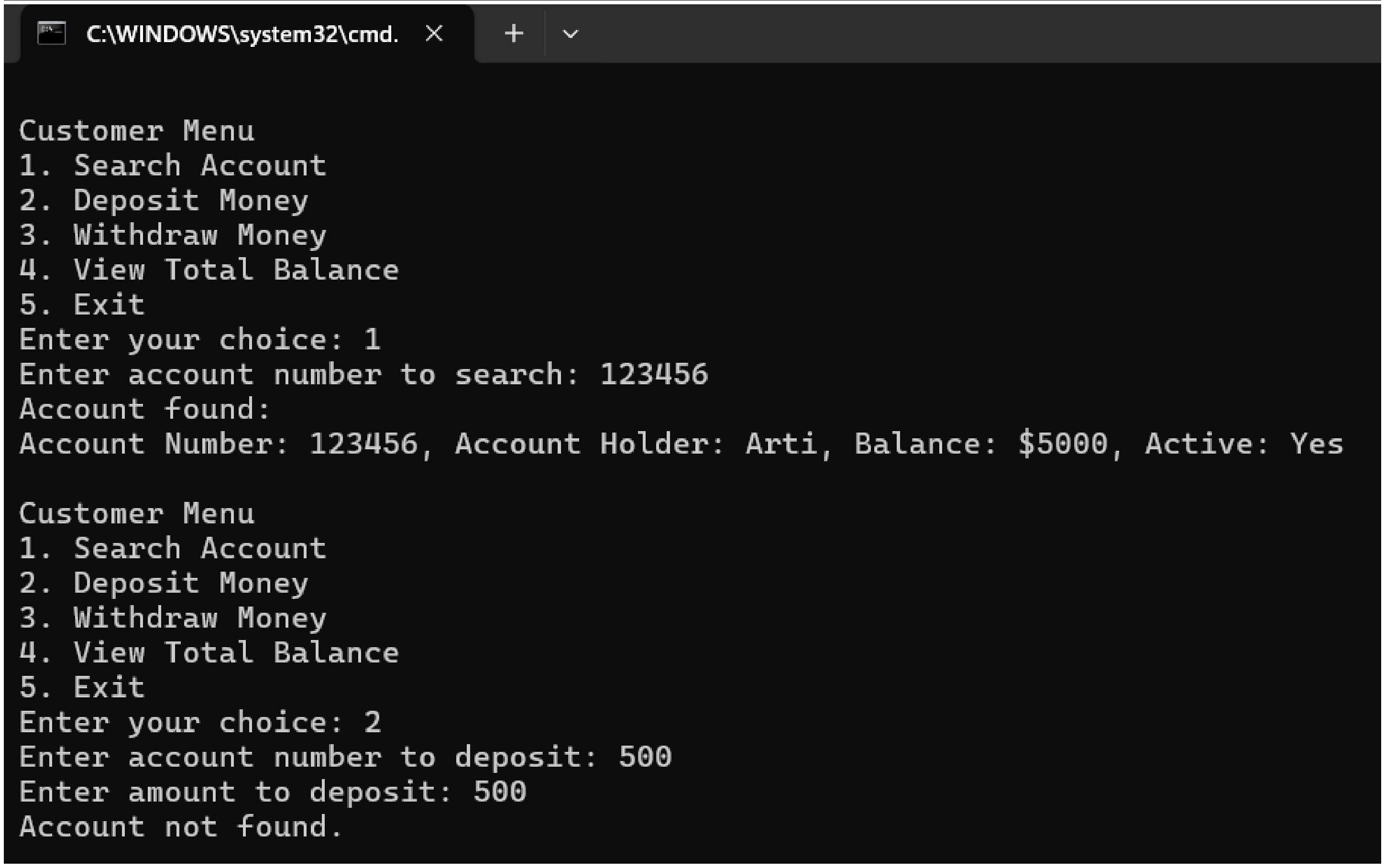
6. **Result/Output/Writing Summary:**



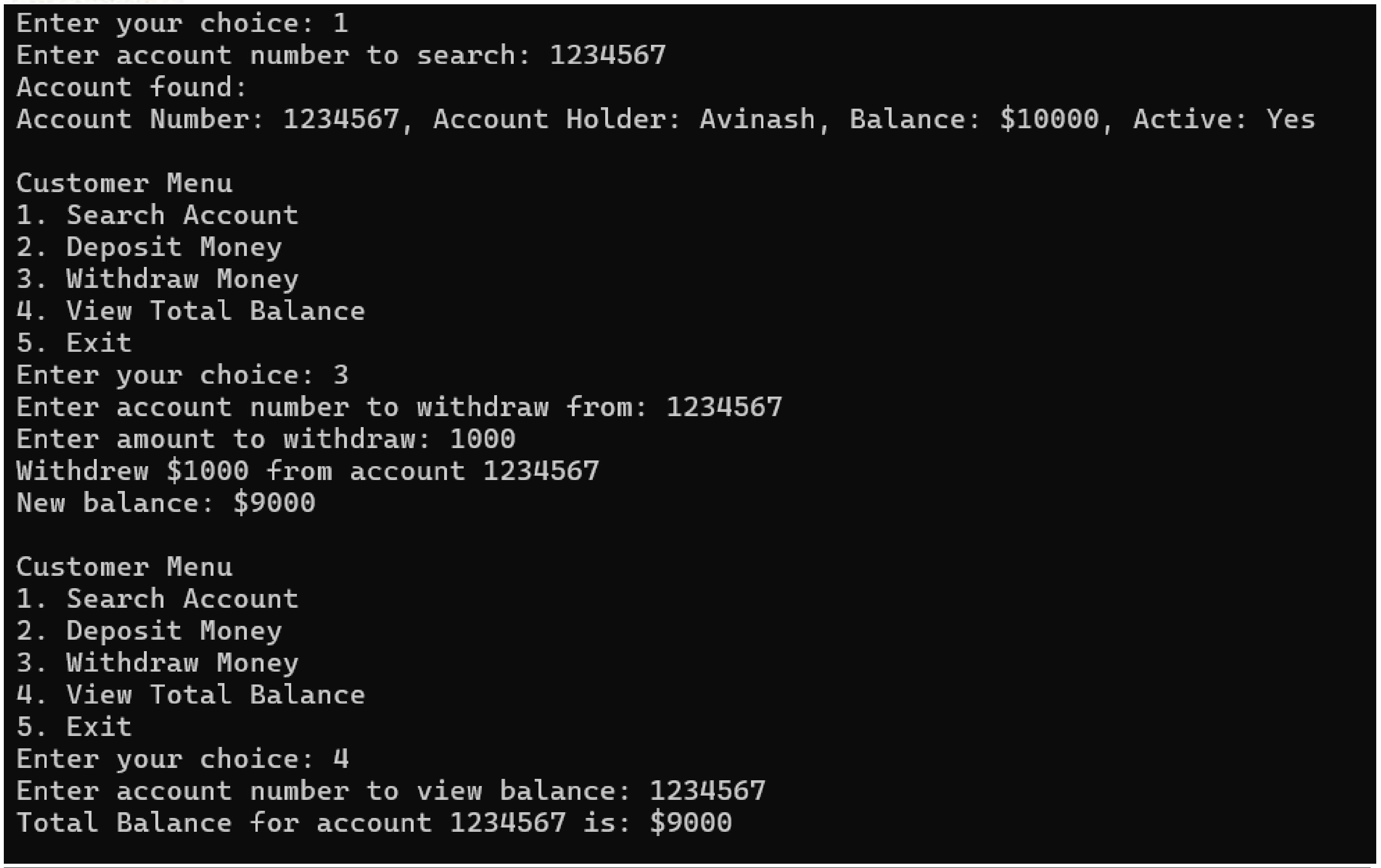
```
Welcome to the Bank Management System
Select User Type:
1. Admin
2. Customer
3. Exit
Enter your choice: 1

Admin Menu
1. Add Account
2. Display Accounts
3. Deactivate Account
4. Activate Account
5. Count Accounts in Bank
6. Exit
Enter your choice: 1
Enter account number: 123456
Enter account holder name: Arti
Enter initial balance: 5000
Account added successfully!

Admin Menu
1. Add Account
2. Display Accounts
3. Deactivate Account
4. Activate Account
5. Count Accounts in Bank
6. Exit
Enter your choice: 1
Enter account number: 1234567
Enter account holder name: Avinash
Enter initial balance: 10000
Account added successfully!
```



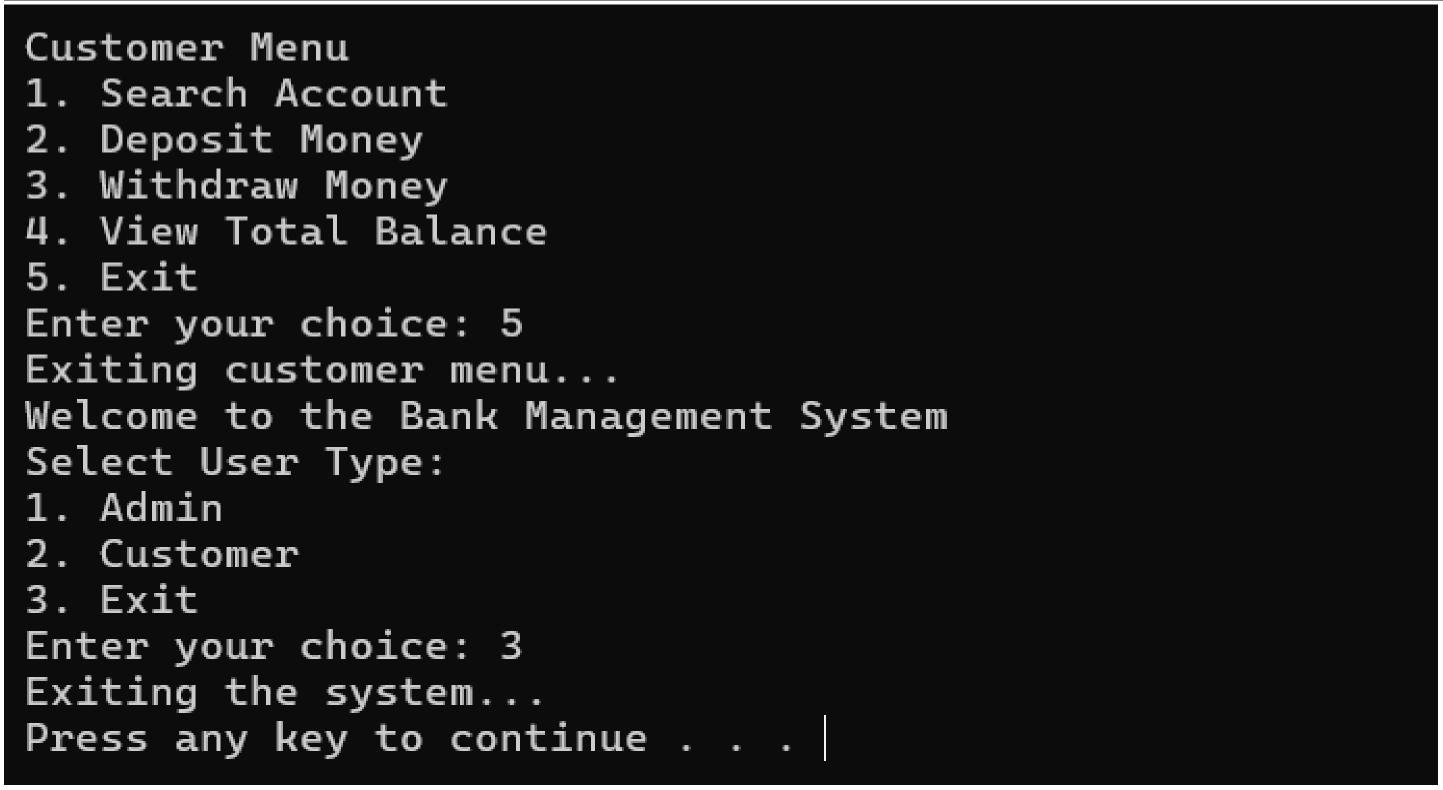
```
4. Activate Account
5. Count Accounts in Bank
6. Exit
Enter your choice: 6
Exiting admin menu...
Welcome to the Bank Management System
Select User Type:
1. Admin
2. Customer
3. Exit
Enter your choice: 2

Customer Menu
1. Search Account
2. Deposit Money
3. Withdraw Money
4. View Total Balance
5. Exit
Enter your choice: 1
Enter account number to search: 123456
Account found:
Account Number: 123456, Account Holder: Arti, Balance: $5000, Active: Yes
```



```

Customer Menu
1. Search Account
2. Deposit Money
3. Withdraw Money
4. View Total Balance
5. Exit
Enter your choice: 1
Enter account number to search: 123456
Account found:
Account Number: 123456, Account Holder: Arti, Balance: $5000, Active: Yes

Customer Menu
1. Search Account
2. Deposit Money
3. Withdraw Money
4. View Total Balance
5. Exit
Enter your choice: 2
Enter account number to deposit: 500
Enter amount to deposit: 500
Account not found.
```



```
Enter your choice: 1
Enter account number to search: 1234567
Account found:
Account Number: 1234567, Account Holder: Avinash, Balance: $10000, Active: Yes

Customer Menu
1. Search Account
2. Deposit Money
3. Withdraw Money
4. View Total Balance
5. Exit
Enter your choice: 3
Enter account number to withdraw from: 1234567
Enter amount to withdraw: 1000
Withdrew $1000 from account 1234567
New balance: $9000

Customer Menu
1. Search Account
2. Deposit Money
3. Withdraw Money
4. View Total Balance
5. Exit
Enter your choice: 4
Enter account number to view balance: 1234567
Total Balance for account 1234567 is: $9000
```



```
Customer Menu
1. Search Account
2. Deposit Money
3. Withdraw Money
4. View Total Balance
5. Exit
Enter your choice: 5
Exiting customer menu...
Welcome to the Bank Management System
Select User Type:
1. Admin
2. Customer
3. Exit
Enter your choice: 3
Exiting the system...
Press any key to continue . . .
```

Code Explanation

This code defines a simple Bank Management System in C# with a console-based interface. It simulates basic banking operations such as adding accounts, searching for accounts, depositing, withdrawing, viewing balance, activating and deactivating accounts. The code is organized into classes and functions to manage both administrator and customer interactions. Below is a breakdown of each part:

1. Class Definitions

a) Account Class

- This class represents a single bank account.
- **Properties:**
  - **AccountNumber:** Stores the account number as a string.
  - **AccountHolderName:** Stores the account holder's name.
  - **Balance:** Stores the account's balance as a floating-point number.
  - **IsActive:** A boolean indicating whether the account is active or inactive (default is true).
- **Constructor:**
  - Initializes an account with an account number, account holder name, and balance.
  - By default, the account is set to active.

b) Bank Class

- This class manages a collection of Account objects and provides methods for various banking operations.
- **Constants and Fields:**
  - **MAX_ACCOUNTS:** The maximum number of accounts allowed (100 in this case).
  - **accounts:** A list of Account objects to store bank accounts.

- **count:** Tracks the current number of accounts in the bank.

- **Methods:**
    - **AddAccount:** Adds a new account to the bank if the maximum limit is not reached.
    - **DisplayAccounts:** Displays all accounts with details like account number, holder name, balance, and active status.
    - **SearchAccount:** Searches for an account by its account number and displays its details.
    - **ViewTotalBalance:** Shows the total balance of a specified account.
    - **DepositMoney:** Deposits an amount into an account if it is active.
    - **WithdrawMoney:** Withdraws an amount from an account if it is active and has a sufficient balance.
    - **DeactivateAccount:** Sets an account's status to inactive.
    - **ActivateAccount:** Sets an account's status to active.
    - **GetAccountCount:** Returns the total number of accounts in the bank.

2. Program Class (Main Application)

This class contains the main program logic for interacting with the user through a console-based menu system.

a) Admin Menu (AdminMenu method)

- This menu provides options specific to the administrator:
    - **Option 1:** Add an account by entering the account number, holder name, and initial balance.
    - **Option 2:** Display all accounts in the bank.
    - **Option 3:** Deactivate an account by entering the account number.
    - **Option 4:** Activate an account by entering the account number.
    - **Option 5:** View the total number of accounts in the bank.

- **Option 6:** Exit the admin menu.

b) Customer Menu (CustomerMenu method)

- This menu provides options specific to customers:
  - **Option 1:** Search for an account by its account number.
  - **Option 2:** Deposit money into an account by specifying the account number and amount.
  - **Option 3:** Withdraw money from an account by specifying the account number and amount.
  - **Option 4:** View the total balance of an account.
  - **Option 5:** Exit the customer menu.

c) Main Program Logic (Main method)

- The main method initializes a Bank object and prompts the user to select their role (Admin or Customer).
- **User Type Selection:**
  - **Admin (1):** Calls the AdminMenu method to interact with the administrator features.
  - **Customer (2):** Calls the CustomerMenu method to interact with the customer features.
  - **Exit (3):** Exits the application.

Summary of Functional Flow

1. **User Role Selection:**
   - User selects either Admin or Customer.
2. **Admin Functions:**
   - Can add, activate/deactivate accounts, display all accounts, and check the total number of accounts.
3. **Customer Functions:**

- Can search for accounts, deposit, withdraw money, and view balances.

4. **Exit:**
   - The application terminates when the user chooses to exit.

Learning outcomes (What I have learnt):

1. **Understanding Object-Oriented Programming (OOP) Principles**
   - Learn how to design and implement classes and objects in a banking context, including encapsulation, data hiding, and the use of constructors to initialize account states.
2. **Data Management Skills**
   - Gain experience in managing collections of data (e.g., lists of bank accounts) and performing operations such as adding accounts, searching for accounts, and updating balances efficiently.
3. **User Interface Design**
   - Develop skills in creating user-friendly command-line interfaces for bank management, facilitating user interactions such as account activation, deposits, withdrawals, and viewing balances, while providing clear feedback.
4. **Problem-Solving and Logic Development**
   - Enhance problem-solving abilities by designing algorithms to manage banking operations, such as depositing, withdrawing, activating/deactivating accounts, and implementing search functionalities.
5. **Testing and Debugging Practices**
   - Understand the importance of testing various banking features, identifying bugs, and applying debugging techniques to ensure the application operates as intended, including handling invalid inputs and edge cases.

Evaluation Grid:

Sr. No.	Parameters	Marks Obtained	Maximum Marks
1.	Demonstration and Performance (Pre Lab Quiz)		5
2.	Worksheet		10
3.	Post Lab Quiz		5