Task 1: Conditional Statements Control Structure In a bank, you have been given the task is to create a program that checks if a customer is eligible for a loan based on their credit score and income. The eligibility criteria are as follows: • Credit Score must be above 700. • Annual Income must be at least $50,000. Tasks: 1. Write a program that takes the customer's credit score and annual income as input. 2. Use conditional statements (if-else) to determine if the customer is eligible for a loan. 3. Display an appropriate message based on eligibility.

```
class Program
{
    static void Main()
    {  loanEligibility();
        ATMTransaction();
        calculateCompoundInterest();
        accountBalanceCheck();
        passwordValidation();
        bankTransactions();
    }
    static void loanEligibility()
    {
        Console.Write("enter credit score: ");
        int creditScore = int.Parse(Console.ReadLine());
        Console.Write("enter annual income: ");
        double annualIncome = double.Parse(Console.ReadLine());

        if (creditScore > 700 && annualIncome >= 50000)
            Console.WriteLine("loan approved");
        else
            Console.WriteLine("loan denied");
    }
```

Task 2: Nested Conditional Statements Create a program that simulates an ATM transaction. Display options such as "Check Balance," "Withdraw," "Deposit,". Ask the user to enter their current balance and the amount they want to withdraw or deposit. Implement checks to ensure that the withdrawal amount is not greater than the available

balance and that the withdrawal amount is in multiples of 100 or 500. Display appropriate messages for success or failure.

```
static void ATMTransaction()
{
    Console.Write("enter current balance: ");
    double balance = double.Parse(Console.ReadLine());

    Console.WriteLine("select an option: 1-check Balance, 2-withdraw, 3-deposit");
    int option = int.Parse(Console.ReadLine());

    if (option == 1)
    {
        Console.WriteLine($"your balance is: {balance}");
    }
    else if (option == 2)
    {
        Console.Write("enter withdrawal amount: ");
        double amount = double.Parse(Console.ReadLine());
        if (amount > balance)
            Console.WriteLine("insufficient funds");
        else if (amount % 100 == 0 || amount % 500 == 0)
            Console.WriteLine($"Withdrawal successful. Remaining balance: {balance -
amount}");
        else
            Console.WriteLine("Amount must be in multiples of 100 or 500");
    }
    else if (option == 3)
    {
        Console.Write("enter deposit amount: ");
```

```
        double deposit = double.Parse(Console.ReadLine());
        balance += deposit;
        Console.WriteLine($"deposit successful. New balance: {balance}");
    }
}
```

Task 3: Loop Structures You are responsible for calculating compound interest on savings accounts for bank customers. You need to calculate the future balance for each customer's savings account after a certain number of years. Tasks: 1. Create a program that calculates the future balance of a savings account. 2. Use a loop structure (e.g., for loop) to calculate the balance for multiple customers. 3. Prompt the user to enter the initial balance, annual interest rate, and the number of years. 4. Calculate the future balance using the formula: future_balance = initial_balance * (1 + annual_interest_rate/100)^years. 5. Display the future balance for each customer.

```
static void CalculateCompoundInterest()
{
    Console.Write("enter initial balance: ");
    double initialBalance = double.Parse(Console.ReadLine());
    Console.Write("enter annual interest rate (%): ");
    double rate = double.Parse(Console.ReadLine());
    Console.Write("enter number of years: ");
    int years = int.Parse(Console.ReadLine());

    double futureBalance = initialBalance * Math.Pow(1 + rate / 100, years);
    Console.WriteLine($"Future Balance: {futureBalance}");
}
```

Task 4: Looping, Array and Data Validation You are tasked with creating a program that allows bank customers to check their account balances. The program should handle multiple customer accounts, and the customer should be able to enter their account

number, balance to check the balance. Account Number should be in the format (first four lettersshould be INDB followedBY 4 numbers , Eg INDB2345) Tasks: 1. Create a C# program that simulates a bank with multiple customer accounts. 2. Use a loop (e.g., while loop) to repeatedly ask the user for their account number and balance until they enter a valid account number. 3. Validate the account number entered by the user. 4. If the account number is valid, display the account balance. If not, ask the user to try again.

```
static void AccountBalanceCheck()
{
    string[] accountNumbers = { "INDB2345", "INDB6789" };
    double[] balances = { 15000, 25000 };

    while (true)
    {
        Console.Write("enter account number (e.g., INDB1234): ");
        string accountNumber = Console.ReadLine();

        if (accountNumber.Length != 8 || !accountNumber.StartsWith("INDB"))
        {
            Console.WriteLine("Invalid format. Account number should start with 'INDB' and be 8 characters long.");
            continue;
        }

        bool isValid = true;
        for (int i = 4; i < 8; i++)
        {
            if (!char.IsDigit(accountNumber[i]))
            {
                isValid = false;
                break;
            }
```

```
        }

        if (!isValid)
        {
            Console.WriteLine("Invalid format. The last four characters must be numbers.");
            continue;
        }

        bool found = false;
        for (int i = 0; i < accountNumbers.Length; i++)
        {
            if (accountNumbers[i] == accountNumber)
            {
                Console.WriteLine($"Account Balance: {balances[i]}");
                found = true;
                break;
            }
        }

        if (found)
            break;
        else
            Console.WriteLine("Account not found. Try again.");
    }
}
```

Task 5: Password Validation Write a program that prompts the user to create a password for their bank account. Implement if conditions to validate the password according to these rules: • The password must be at least 8 characters long. • It must contain at least one uppercase letter. • It must contain at least one digit. • Display appropriate messages to indicate whether their password is valid or not.

```
static void PasswordValidation()
{
    Console.Write("Create Password: ");
    string password = Console.ReadLine();

    if (password.Length >= 8 &&
        Regex.IsMatch(password, "[A-Z]") &&
        Regex.IsMatch(password, "\d"))
    {
        Console.WriteLine("Password is valid");
    }
    else
    {
        Console.WriteLine("Password must be at least 8 characters long, contain an uppercase letter, and a digit");
    }
}
```

Task 6: Create a program that maintains a list of bank transactions (deposits and withdrawals) for a customer. Use a while loop to allow the user to keep adding transactions until they choose to exit. Display the transaction history upon exit using looping statements.

```
static void BankTransactions()
{
    List<string> transactions = new List<string>();
    double balance = 0;
    string choice;
    do
    {
        Console.WriteLine("Enter transaction: 1-Deposit, 2-Withdraw, 3-Exit");
```

```
    choice = Console.ReadLine();

    if (choice == "1")
    {
        Console.Write("Enter deposit amount: ");
        double deposit = double.Parse(Console.ReadLine());
        balance += deposit;
        transactions.Add($"Deposited: {deposit}, New Balance: {balance}");
    }
    else if (choice == "2")
    {
        Console.Write("Enter withdrawal amount: ");
        double withdraw = double.Parse(Console.ReadLine());
        if (withdraw > balance)
        {
            Console.WriteLine("Insufficient funds");
        }
        else
        {
            balance -= withdraw;
            transactions.Add($"Withdrawn: {withdraw}, New Balance: {balance}");
        }
    }
} while (choice != "3");

Console.WriteLine("Transaction History:");
foreach (var transaction in transactions)
{
    Console.WriteLine(transaction);
```

```
        }
    }
}
```

----------day 2

Task 7: Class & Object 1. Create a Customer class with the following confidential attributes: • Attributes o Customer ID o First Name o Last Name o Email Address o Phone Number o Address • Constructor and Methods o Implement default constructors and overload the constructor with Customer attributes, generate getter and setter, (print all information of attribute) methods for the attributes. 2. Create an Account class with the following confidential attributes: • Attributes o Account Number o Account Type (e.g., Savings, Current) o Account Balance • Constructor and Methods o Implement default constructors and overload the constructor with Account attributes, o Generate getter and setter, (print all information of attribute) methods for the attributes. o Add methods to the Account class to allow deposits and withdrawals. - deposit(amount: float): Deposit the specified amount into the account. withdraw(amount: float): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance. - calculate_interest(): method for calculating interest amount for the available balance. interest rate is fixed to 4.5% • Create a Bank class to represent the banking system. Perform the following operation in main method: o create object for account class by calling parameter constructor. o deposit(amount: float): Deposit the specified amount into the account. o withdraw(amount: float): Withdraw the specified amount from the account. o calculate_interest(): Calculate and add interest to the account balance for savings accounts.

```
class Customer
{
    private string customerID;
    private string firstName;
    private string lastName;
    private string email;
    private string phoneNumber;
    private string address;
```

```
    public Customer() { }

    public Customer(string customerID, string firstName, string lastName, string email, string phoneNumber, string address)
    {
        this.customerID = customerID;
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
        this.phoneNumber = phoneNumber;
        this.address = address;
    }

    public string GetCustomerID() { return customerID; }
    public void SetCustomerID(string customerID) { this.customerID = customerID; }

    public string GetFirstName() { return firstName; }
    public void SetFirstName(string firstName) { this.firstName = firstName; }

    public string GetLastName() { return lastName; }
    public void SetLastName(string lastName) { this.lastName = lastName; }

    public string GetEmail() { return email; }
    public void SetEmail(string email) { this.email = email; }

    public string GetPhoneNumber() { return phoneNumber; }
    public void SetPhoneNumber(string phoneNumber) { this.phoneNumber = phoneNumber; }
```

```
    public string GetAddress() { return address; }
    public void SetAddress(string address) { this.address = address; }

    public void PrintCustomerInfo()
    {
        Console.WriteLine($"Customer ID: {customerID}\nName: {firstName} {lastName}\nEmail: {email}\nPhone: {phoneNumber}\nAddress: {address}");
    }
}
```

```
class Account
{
    private string accountNumber;
    private string accountType;
    private double balance;

    public Account() { }

    public Account(string accountNumber, string accountType, double balance)
    {
        this.accountNumber = accountNumber;
        this.accountType = accountType;
        this.balance = balance;
    }

    public string GetAccountNumber() { return accountNumber; }
    public void SetAccountNumber(string accountNumber) { this.accountNumber = accountNumber; }
```

```
public string GetAccountType() { return accountType; }
public void SetAccountType(string accountType) { this.accountType = accountType; }

public double GetBalance() { return balance; }
public void SetBalance(double balance) { this.balance = balance; }

public void Deposit(double amount)
{
   balance += amount;
   Console.WriteLine($"Deposited: {amount}, New Balance: {balance}");
}

public void Withdraw(double amount)
{
   if (amount > balance)
      Console.WriteLine("Insufficient balance!");
   else
   {
      balance -= amount;
      Console.WriteLine($"Withdrawn: {amount}, Remaining Balance: {balance}");
   }
}

public void CalculateInterest()
{
   if (accountType == "Savings")
   {
      double interest = balance * 0.045;
      balance += interest;
```

```
            Console.WriteLine($"Interest Added: {interest}, New Balance: {balance}");
        }
    }

    public void PrintAccountInfo()
    {
        Console.WriteLine($"AccountNumber: {accountNumber}\nType: {accountType}\nBalance: {balance}");
    }
}
```

```
class Bank
{
    static void Main(string[] args)
    {
        Account acc = new Account("1001", "Savings", 5000);
        acc.PrintAccountInfo();

        acc.Deposit(2000);
        acc.Withdraw(1000);
        acc.CalculateInterest();
        acc.PrintAccountInfo();
    }
}
```