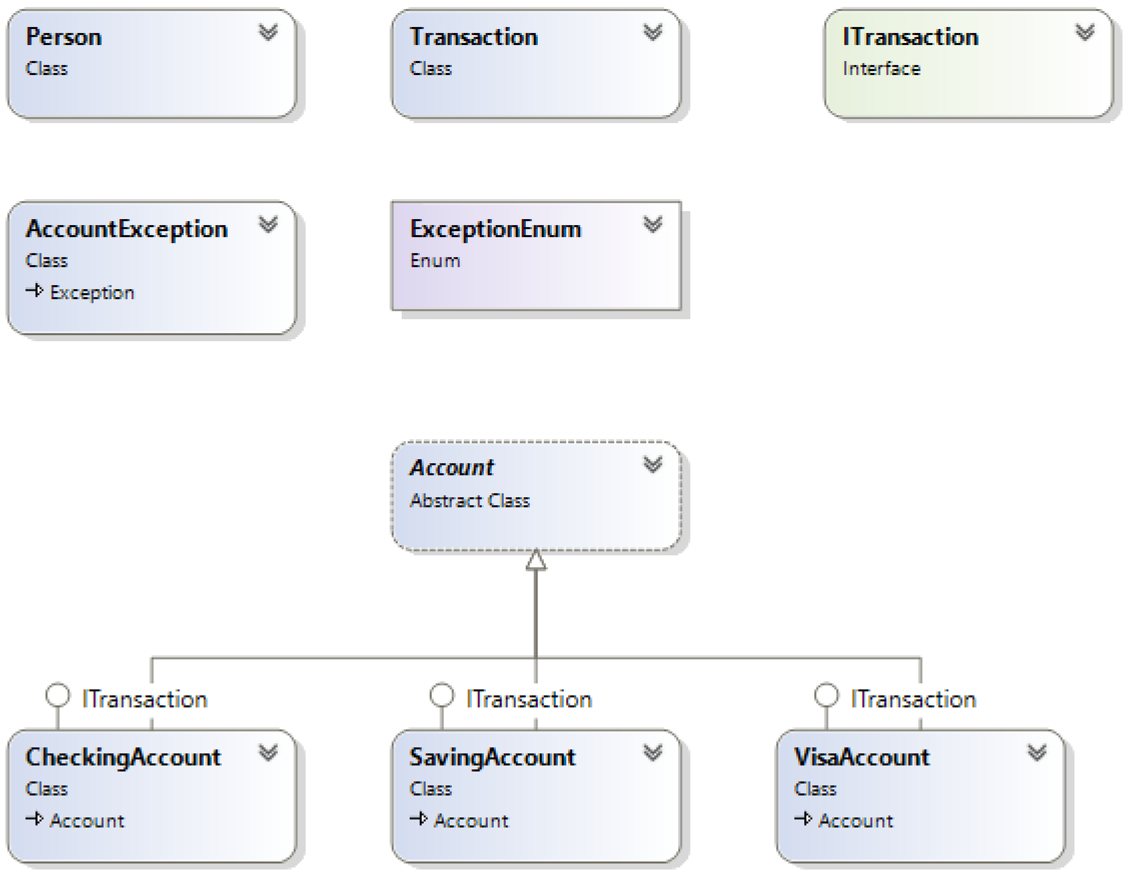
Programming III

Final Exam, – Implementing a Banking Application

Due: Upload zipped solution folder to e-centennial by Sunday 9.00 am , December 10th

Upload a word document with the screenshots of the output. (Not the screenshots of source code, screen shots of the output)



The application is by far the most complex system that you have attempted so far. It consists of nine classes one interface and one enum linked in varying degrees of tightness. The Bank class is the main driver of the application. It has a collection of Accounts and Person that is initialize appropriately in the static constructor. You will implement the entire system in Visual Studio. A short description of each types with their members is given below.

You are advised to implement each type in the order that they are described.

Each type must be in separate files.

Person class

You will implement the Person Class in Visual Studio. A person object may be associated with multiple accounts. A person initiates activities (deposits or withdrawal) against an account that is captured in a transaction object.

A short description of each class member is given below:

Person Class
Fields
- password : string
Properties
+ «C# property, setter private» IsAuthenticated : bool + «C# property, setter absent» SIN : string + «C# property, setter absent» Name : string
Methods
+ «Constructor» Person(name : string, sin : string) + Login(password : string) : void + Logout() : void + ToString() : string

Fields:

1. **`password`** – this private **`string`** field represents the password of this person. (N.B. Password are not normally stored as text but as a hash value. A hashing function operates on the password and the result is stored in the field. When a user supplies a password it is passed through the same hashing function and the result is compared to the value of the field.).

Properties:

1. **`SIN`** – this **`string`** property represents the sin number of this person. This getter is public and setter is absent.
2. **`IsAuthenticated`** – this property is a **`bool`** representing if this person is logged in with the correct password. This is modified in the **`Login()`** and the **`Logout()`** methods. This is an auto-implemented property, and the getter is public and setter is private
3. **`Name`** – this property is a **`string`** representing the name of this person. The getter is public and setter is absent.

Methods:

1. `public Person( string name, string sin )` – This public constructor takes two parameters: a string representing the name of the person and another string representing the SIN of this person. It does the following:
   a. The method assigns the arguments to the appropriate fields.
   b. It also sets the password to the first three letters of the SIN. [use the `Substring(start_position, length)` method of the string class]
2. `public void Login( string password )` – This method takes a string parameter representing the password and does the following:
   a. If the argument DOES NOT match the password field, it does the following:
      - Sets the `IsAuthenticated` property to `false`
      - Creates an `AccountException` object using argument `AccountEnum.PASSWORD_INCORRECT`
      - Throws the above exception
   b. If the argument matches the password, it does the following:
      - Sets the `IsAuthenticated` property is set to `true`

   This method does not display anything
3. `public void Logout( )` – This is public method does not take any parameters nor does it return a value.
   This method sets the `IsAuthenticated` property to `false`
   This method does not display anything
4. `public override string ToString( )`– This public method overrides the same method of the Object class. It returns a string representing the name of the person~~ and if he is authenticated or not~~.

ITransaction interface

You will implement the `ITransaction` Interface in Visual Studio. The three sub classes implement this interface.

A short description of each class member is given below:

`ITransaction` Interface
Methods

`Withdraw(amount : double, person : Person) : void`
`Deposit(amount : double, person : Person) : void`

Methods:

1. `void Withdraw(double amount, Person person).`
2. `void Deposit(double amount, Person person).`

Transaction class

You will implement the Transaction Class in Visual Studio. The only purpose of this class is to capture the data values for each transaction. A short description of the class members is given below:

All the properties are public with the setter absent.

Transaction Class
Properties
✚ «C# property, setter absent » AccountNumber : string ✚ «C# property, setter absent» Amount : double ✚ «C# property, setter absent» Originator : Person ✚ «C# property, setter absent» Time : DateTime
Methods
✚ «Constructor» Transaction(accountNumber : string, amount : double, endBalance : double, person : Person, time : DateTime) ✚ ToString() : string

Properties:

All the properties are public with the setter absent

1. **`AccountNumber`** – this property is a **`string`** representing the account number associated with this transaction. The getter is public and the setter is absent.
2. **`Amount`** – this property is a **`double`** representing the account of this transaction. The getter is public and the setter is absent.
3. **`Originator`** – this property is a `Person` representing the person initiating this transaction. The getter is public and the setter is absent.
4. **`Time`** – this property is a `DateTime` (a .NET built-in type) representing the time associated with this transaction. The getter is public and the setter is absent.

Methods:

1. **`public Transaction( string accountNumber, double amount, Person person, DateTime time )`** – This public constructor takes five arguments. It assigns the arguments to the appropriate fields.
2. **`public override string ToString( )`** – This method overrides the same method of the Object class. It does not take any parameter and it returns a string representing the account number, name of the person, the amount and the time that this transition was completed. [you may use **`ToShortTimeString()`** method of the **`DateTime`** class]. You **must** include the word Deposit or Withdraw in the output.

A better type would have been a struct instead of a class.

ExceptionEnum enum

You will implement this enum in Visual Studio. There are seven members:

```
ACCOUNT_DOES_NOT_EXIST,
CREDIT_LIMIT_HAS_BEEN_EXCEEDED,
NAME_NOT_ASSOCIATED_WITH_ACCOUNT,
NO_OVERDRAFT,
PASSWORD_INCORRECT,
USER_DOES_NOT_EXIST,
USER_NOT_LOGGED_IN
```

The members are self-explanatory.

AccountException class

You will implement the AccountException Class in Visual Studio. This inherits from the `Exception` Class to provide a custom exception object for this application. It consists of seven strings and two constructors:

AccountException Class → Exception
Fields
Properties
Methods
+ «Constructor» AccountException(reason : ExceptionEnum)

Fields:

There are no fields

Properties:

There are no properties

Methods:

1. **`AccountException( ExceptionEnum reason )`**– this public constructor simply invokes the base constructor with an appropriate argument. Use the `ToString()` of the enum type.

Account class

You will implement the Account Class in Visual Studio. This class that will serve as the base class for the Visa, Checking and Saving classes. The member `PrepareMonthlyStatement()` is abstract so the class must be declared abstract. Because this is an abstract class, you may not instantiate this class. A short description of the class members is given below:

Account Abstract Class
Fields
«readonly» users : List<Person> **#** «readonly» transactions : List<Transaction> **$-** LAST_NUMBER = 100_000: int
Properties
+ «C# property, setter absent» Number: string **+** «C# property, protected setter» Balance: double **+** «C# property, protected setter» LowestBalance : double
Methods
+ «Constructor» Account(type : string, balance : double) **+** Deposit(balance : double, person : Person) : void **+** AddUser(person : Person) : string **+** IsUser(name : string) : bool **+** «C# abstract method» PrepareMonthlyStatement() : void **+** ToString() : string

Fields:

If these two fields are `readonly`, then how it is possible to add users and transactions?

1. **`users`** – this field is a list of persons representing the user who have access to this account. This is `readonly` and public and is initialized at declaration.
2. **`transactions`** – this field is a list of transaction representing the deposits, withdrawal, payments and purchases of this account. This is `readonly` and public and is initialized at declaration.
3. **`LAST_NUMBER`** – this private field is a class variable of type int. It represents the last account number that was used to generate the unique account number. It is initialized at declaration to `100,000`

Properties:

1. **`Balance`** – this property is a double representing the amount in this account. This is modified in the **`Deposit()`** method and in the **`PrepareMonthlyReport()`** method of the derived classes. This is an auto-implemented property the getter is public and the setter is protected
2. **`LowestBalance`** – this property is a double representing the lowest balance that this amount ever drops to. This is modified in the **`Deposit()`** method. This is an auto-implemented property the getter is public and the setter is protected
3. **`Number`** – this property is a string representing the account number of this account. The getter is public and the setter is absent.

Methods:

1. **`public Account( string type, double balance )`** – This is the public constructor. It takes two parameters: a three-letter string representing the type of the account **(“`VS-`”**, **“`SV-`”** and **“`CK-`”** for visa, saving and checking account respectively) and a double representing the starting balance. The method does the following:
    a. Sets the `Number` property to the concatenation (joining) of the first argument and the class variable `LastNumber`
    b. Increments the class variable `LastNumber`
    c. Assigns the second argument to the property `Balance`
    d. Assigns the second argument to the property `LowestBalance`

   This constructor is called in the constructors on the three derived classes
2. **`public void Deposit( double amount, Person person )`** – This public method take a double parameter representing the amount to change balance by and a person object representing the person who is performing this transaction. This method does the following:
    a. Increase (or decrease) the property `Balance` by the amount specified by its argument.
    b. Update the property `LowestBalance` based on the current value of Balance.
    c. Create a Transaction object based on the current time (use DateTime.Now), the AccountNumber, the amount (specified by the argument), a person object (as specified by the argument.
    d. Adds the above object to the list of **`transactions`**

   This is method is called by the **`Deposit`** and **`Withdraw`** methods of the derived classes **`CheckingAccount`** and **`SavingsAccount`** as well as the **`DoPurchase`** and **`DoPayment`** of the **`VisaAccount`** class.
   This method does not display anything nor does it return a value.
3. **`public void AddUser( Person person )`** – This public method takes a person object as a parameter. It adds the argument to holders (the list of persons). This method does not return a value nor does it display anything on the screen.

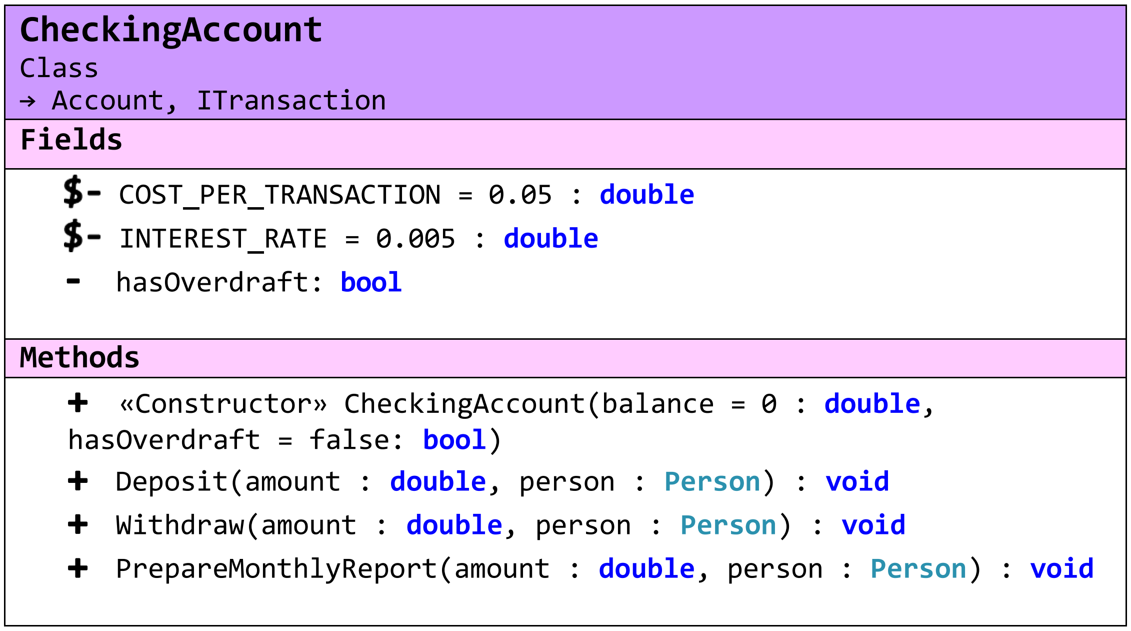
4. `public bool IsUser( string name )` – This public method that takes a string parameter representing the name of the user and returns `true` if the argument matches the name of a person in holders (the list of persons) and `false` otherwise.

   Note: This is not a sound design because it is possible that multiple users will have the same name

   You cannot use the Contains method of the list class because the list is a list of persons and not a list of strings. You will have to use a loop to check each person in the collection. It does not display anything on screen
5. *`public abstract void PrepareMonthlyReport( )`– This abstract public method does not take any parameter nor does it return a value.*
   Research how to declare an abstract method.
   This method is implemented in the classes derived from `Accounts`.
6. `public override string ToString( )` – This method overrides the same method of the Object class. It does not take any parameter but returns a string representation of this account. It does the following:
   a. Declare and initialise a string variable to store the return value and add the following to it:
      - The Number of this account
      - The names of each of the users of the account
      - The Balance
      - All the transactions

CheckingAccount class

You will implement the CheckingAccount Class in Visual Studio. This is a sub class is derived from the Account class and implements the ITransaction interface. There are two class variables i.e. variables that are shared but all the objects of this class. A short description of the class members is given below:



```
CheckingAccount
Class
→ Account, ITransaction
Fields
    $- COST_PER_TRANSACTION = 0.05 : double
    $- INTEREST_RATE = 0.005 : double
    -  hasOverdraft: bool
Methods
    +  «Constructor» CheckingAccount(balance = 0 : double,
    hasOverdraft = false: bool)
    +  Deposit(amount : double, person : Person) : void
    +  Withdraw(amount : double, person : Person) : void
    +  PrepareMonthlyReport(amount : double, person : Person) : void
```

Fields:

1. **`COST_PER_TRANSACTION`** – this is a class variable of type double representing the unit cost per transaction. All of the objects on this class will have the same value. This class variable is initialized to **`0.05`**.
2. **`INTEREST_RATE`** – this is a class variable of type double representing the annual interest rate. All of the objects on this class will have the same value. This class variable is initialized to **`0.005`**.
3. **`hasOverdraft`** – this is a bool indicating if the balance on this account can be less than zero. This private instance variable is set in the constructor.

Methods:

1. **`public CheckingAccount( double balance = 0, bool hasOverdraft = false )`** – This public constructor takes a parameter of type double representing the starting balance of the account and a bool indicating if this account has over draft permission. The constructor does the following:
    a. It invokes the base constructor with the string "CK-" and the appropriate argument.

   b. Assigns the `hasOverdraft` argument to the appropriate field.

2. **`public new void Deposit( double amount, Person person )`** – this public method takes two arguments: a double representing the amount to be deposited and a person object representing the person do the transaction. The method does the following:
   a. Calls the **`Deposit()`** method of the base class with the appropriate arguments

This definition hides the corresponding member in the parent class because the base class implementation is needed for this method as well as the `Withdraw()` method

3. **`public void Withdraw( double amount, Person person )`** – this public method takes two arguments: a double representing the amount to be withdrawn and a person object representing the person do the transaction. The method does the following:
   a. Throws an **`AccountException`** object if this person in not associated with this account.
   b. Throws an **`AccountException`** object if this person in not logged in.
   c. Throws an **`AccountException`** object if the withdrawal amount is greater than the balance and there is no overdraft facility.
   d. Otherwise it calls the **`Deposit()`** method of the base class with the appropriate arguments (you will send negative of the amount)

4. **`public override void PrepareMonthlyReport( )`** – this public method override the method of the base class with the same name. The method does the following:
   a. Calculate the service charge by multiplying the number of transactions by the **`COST_PER_TRANSACTION`** (how can you find out the number of transactions?)
   b. Calculate the interest by multiplying the **`LowestBalance`** by the **`INTEREST_RATE`** and then dividing by 12
   c. Update the **`Balance`** by adding the interest and subtracting the service charge
   d. **`transactions`** is re-initialized (use the **`Clear()`** method of the list class)

   This method does not take any parameter nor does it display anything

In a real-world application, the transaction objects would be archived before clearing.

SavingAccount class

You will implement the SavingAccount Class in Visual Studio. This is a sub class derived from the `Account` class and implements the `ITransaction` interface. Again, there are two class variables. A short description of the class members is given below:

```
SavingAccount
Class
→ Account, ITransaction
Fields
    $- COST_PER_TRANSACTION : double
    $- INTEREST_RATE : double
Methods
    + «Constructor» SavingAccount(balance = 0 : double)
    + Deposit(amount : double, person : Person) : void
    + Withdraw(amount : double, person : Person) : void
    + PrepareMonthlyReport(amount : double, person : Person) : void
```

Fields:

1. **`COST_PER_TRANSACTION`** – this is a class variable of type double representing the unit cost per transaction. All of the objects on this class will have the same value. This class variable is initialized to **`0.05`**.
2. **`INTEREST_RATE`** – this is a class variable of type double representing the annual interest rate. All of the objects on this class will have the same value. This class variable is initialized to **`0.015`**.

Methods:

1. **`public SavingAccount( double balance = 0 )`** – This public constructor takes a parameter of type double representing the starting balance of the account. The constructor does the following:
    a. It invokes the base constructor with the string "SV-" and its argument.
2. **`public new void Deposit( double amount, Person person )`** – this public method takes two arguments: a double representing the amount to be deposited and a person object representing the person do the transaction. The method calls the **`Deposit()`** method of the base class with the appropriate arguments.
3. **`public void Withdraw( double amount, Person person  )`** – this public method takes two arguments: a double representing the amount to be withdrawn and a person object representing the person do the transaction. The method does the following:

  a. Throws an `AccountException` object if this person in not associated with this account.
  b. Throws an `AccountException` object if this person in not logged in.
  c. Throws an `AccountException` object if the intended withdrawal amount exceeds the balance.
  d. Otherwise it calls the **`Deposit()`** method of the base class with the appropriate arguments (you will send negative of the amount)

4. **`public override void PrepareMonthlyReport( )`** – this public method override the method of the base class with the same name. The method does the following:
   a. Calculate the service charge by multiplying the number of transactions by the **`COST_PER_TRANSACTION`** (how can you find out the number of transactions?)
   b. Calculate the interest by multiplying the **`LowestBalance`** by the **`INTEREST_RATE`** and then dividing by 12
   c. Update the **`Balance`** by adding the interest and subtracting the service charge
   d. **`transactions`** is re-initialized (use the **`Clear()`** method of the list class)

   This method does not take any parameters, nor does it display anything

VisaAccount class

You will implement the VisaAccount Class in Visual Studio. This is a sub class derived from the `Account` class and implements `ITransaction` interface. This class has a single class variable. A short description of the class members is given below:

```
VisaAccount
Class
→ Account, ITransaction
Fields
    -   creditLimit : double
    $-  INTEREST_RATE = 0.1995 : double
Methods
    +  «Constructor» VisaAccount(balance = 0 : double, creditLimit =
    1200 : double)
    +  DoPayment(amount : double, person : Person) : void
    +  DoPurchase(amount : double, person : Person) : void
    +  PrepareMonthlyReport(amount : double, person : Person) : void
```

Fields:

1. **`creditLimit`** – this is a double representing the maximum balance allowable on this account. This private instance variable is set in the constructor.
2. **`INTEREST_RATE`** – this is a class variable of type double representing the annual interest rate. All of the objects on this class will have the same value. This class variable is initialized to **`0.1995`**.

Methods:

1. **`public VisaAccount( double balance = 0, double creditLimit = 1200 )`** – This public constructor takes a parameter of type double representing the starting balance of the account and a double representing the credit limit of this account. The constructor does the following:
    a. It invokes the base constructor with the string "VS-" and the appropriate argument
    b. Assigns the argument to the appropriate field
2. **`public void DoPayment( double amount, Person person )`** – this public method takes two arguments: a double representing the amount to be deposited and a person object representing the person do the transaction. The method calls the **`Deposit()`** method of the base class with the appropriate arguments

3. **`public void DoPurchase( double amount, Person person )`** – this public method takes two arguments: a double representing the amount to be withdrawn and a person object representing the person do the transaction. The method does the following:
    a. Throws an `AccountException` object if this person in not associated with this account.
    b. Throws an `AccountException` object if this person in not logged in.
    c. Throws an `AccountException` object if this intended purchase will exceed the credit limit.
    d. Otherwise it calls the **`Deposit()`** method of the base class with the appropriate arguments (you will send negative of the amount)
4. **`public override void PrepareMonthlyReport( )`** – this public method override the method of the base class with the same name. The method does the following:
    a. Calculate the interest by multiplying the **`LowestBalance`** by the **`InterestRate`** and then dividing by 12
    b. Update the Balance by subtraction the interest
    c. Transactions is re-initialized

    This method does not take any parameter nor does it display anything

Bank class

You will implement the Bank Class in Visual Studio. This is a static class where all its members are also static. [All the members of a static class must also be declared static.] A short description of the class members is given below:

```
Bank
Static Class
Fields
   $+ «readonly» ACCOUNTS : List<ITransaction>
   $+ «readonly» USERS : List<Person>

Methods
   $  «Constructor» Bank()
   $+GetAccount(number : string) : ITransaction
   $+GetPerson(name : string) : Person
   $+PrintAccounts() : void
   $+PrintUsers() : void
```

Fields:

1. **ACCOUNTS** – this class variable is a list of account. This private member is initialized in the static constructgoor.
2. **USERS** – this class variable is a list of person. This private member is initialized in the static constructor.

Methods:

1. `static Bank( )` – This static constructor contains the following statements to initialize the users and account stores

```
//initialize the USERS collection
USERS = new List<Person>(){
    new Person("Janani", "1234-5678"),  //0

    new Person("Ilia", "2345-6789"),       //1
    new Person("Tom", "3456-7890"),        //2
    new Person("Syed", "4567-8901"),       //3
    new Person("Arben", "5678-9012"),      //4
    new Person("Patrick", "6789-0123"),    //5
    new Person("Yin", "7890-1234"),        //6
    new Person("Hao", "8901-2345"),        //7
    new Person("Jake", "9012-3456"),       //8
```

```
        new Person("Joanne", "1224-5678"),    //9
        new Person("Nicoletta", "2344-6789"), //10
    };


    //initialize the ACCOUNTS collection
    ACCOUNTS = new List<Account>{
        new VisaAccount(),              //VS-100000
        new  VisaAccount(150,  -500),   //VS-100001
        new  SavingAccount(5000),       //SV-100002
        new SavingAccount(),            //SV-100003
        new  CheckingAccount(2000),     //CK-100004
        new CheckingAccount(1500, true),//CK-100005
        new  VisaAccount(50,  -550),    //VS-100006
        new SavingAccount(1000),       //SV-100007
    };
    //associate users with accounts
    ACCOUNTS[0].AddUser(USERS[0]);
    ACCOUNTS[0].AddUser(USERS[1]);
    ACCOUNTS[0].AddUser(USERS[2]);

    ACCOUNTS[1].AddUser(USERS[3]);
    ACCOUNTS[1].AddUser(USERS[4]);
    ACCOUNTS[1].AddUser(USERS[5]);

    ACCOUNTS[2].AddUser(USERS[6]);
    ACCOUNTS[2].AddUser(USERS[7]);
    ACCOUNTS[2].AddUser(USERS[8]);

    ACCOUNTS[3].AddUser(USERS[9]);
    ACCOUNTS[3].AddUser(USERS[10]);

    ACCOUNTS[4].AddUser(USERS[2]);
    ACCOUNTS[4].AddUser(USERS[4]);
    ACCOUNTS[4].AddUser(USERS[6]);

    ACCOUNTS[5].AddUser(USERS[8]);
    ACCOUNTS[5].AddUser(USERS[10]);

    ACCOUNTS[6].AddUser(USERS[1]);
    ACCOUNTS[6].AddUser(USERS[3]);

    ACCOUNTS[7].AddUser(USERS[5]);
    ACCOUNTS[7].AddUser(USERS[7]);
```

2. **`public static void PrintAccounts( )`** – this public static method displays all the accounts in the accounts collection
3. **`public static void PrintPersons( )`** – this public static method displays all the persons in the persons collection
4. **`public static Person GetPerson( string name )`** – this public static method takes a string representing the name of a person and returns the matching person object. The method does the following:

   a. Using a suitable loop iterate thru the static collection persons.
   b. If the person name matches the argument then this person object is returned
   c. If none of name matches, then an `AccountException` object is thrown

   This method does not display anything on screen

5. `public static Account GetAccount( string number )` – this public static method takes a string representing an account number and returns the matching account. The method does the following:
   a. Using a suitable loop iterate thru the list of account
   b. If the account number matches the argument then return this account
   c. If none of the number matched, then an `AccountException` object is thrown

Testing

Use the following code in your test harness.

```
//testing the visa account
Console.WriteLine("\nAll acounts:");
Bank.PrintAccounts();
Console.WriteLine("\nAll Users:");
Bank.PrintPersons();

Person p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10;
p0 = Bank.GetPerson("Janani");
p1 = Bank.GetPerson("Ilia");
p2 = Bank.GetPerson("Tom");
p3 = Bank.GetPerson("Syed");
p4 = Bank.GetPerson("Arben");
p5 = Bank.GetPerson("Patrick");
p6 = Bank.GetPerson("Yin");
p7 = Bank.GetPerson("Hao");
p8 = Bank.GetPerson("Jake");
p9  =  Bank.GetPerson("Joanne");
p10 = Bank.GetPerson("Nicoletta");

p0.Login("123"); p1.Login("234");
p2.Login("345"); p3.Login("456");
p4.Login("567"); p5.Login("678");
p6.Login("789"); p7.Login("890");
p10.Login("234");p8.Login("901");

//a visa account
VisaAccount a = Bank.GetAccount("VS-100000") as VisaAccount;
a.DoPayment(1500, p0);
a.DoPurchase(200, p1);
a.DoPurchase(25, p2);
a.DoPurchase(15, p0);
a.DoPurchase(39, p1);
a.DoPayment(400, p0);
Console.WriteLine(a);

a = Bank.GetAccount("VS-100001") as VisaAccount;
a.DoPayment(500, p0);
a.DoPurchase(25, p3);
a.DoPurchase(20, p4);
a.DoPurchase(15, p5);
Console.WriteLine(a);

//a saving account
ITransaction b = Bank.GetAccount("SV-100002");
b.Withdraw(300, p6);
b.Withdraw(32.90, p6);
b.Withdraw(50, p7);
b.Withdraw(111.11, p8);
Console.WriteLine(b);

b = (SavingAccount)Bank.GetAccount("SV-100003");
b.Deposit(300, p3);     //ok even though p3 is not a holder
```

```
b.Deposit(32.90, p2);
b.Deposit(50, p5);
b.Withdraw(111.11, p10);
Console.WriteLine(b);

//a checking account
ITransaction c = Bank.GetAccount("CK-100004");
c.Deposit(33.33, p7);
c.Deposit(40.44, p7);
c.Withdraw(150, p2);
c.Withdraw(200, p4);
c.Withdraw(645, p6);
c.Withdraw(350, p6);
Console.WriteLine(c);

c = Bank.GetAccount("CK-100005");
c.Deposit(33.33, p8);
c.Deposit(40.44, p7);
c.Withdraw(450, p10);
c.Withdraw(500, p8);
c.Withdraw(645, p10);
c.Withdraw(850, p10);
Console.WriteLine(c);

a = Bank.GetAccount("VS-100006") as VisaAccount;
a.DoPayment(700, p0);
a.DoPurchase(20, p3);
a.DoPurchase(10, p1);
a.DoPurchase(15, p1);
Console.WriteLine(a);

b = Bank.GetAccount("SV-100007");
b.Deposit(300, p3);     //ok even though p3 is not a holder
b.Deposit(32.90, p2);
b.Deposit(50, p5);
b.Withdraw(111.11, p7);
Console.WriteLine(b);

Console.WriteLine("\n\nExceptions:");
//The following will cause exception
try
{
    p8.Login("911");//incorrect password
}
catch (AccountException e) { Console.WriteLine(e.Message); }

try
{
    p3.Logout();
    a.DoPurchase(12.5, p3);     //exception user is not logged in
}
catch (AccountException e) { Console.WriteLine(e.Message); }

try
{
    a.DoPurchase(12.5, p0); //user is not associated with this account
}
catch (AccountException e) { Console.WriteLine(e.Message); }
```

```
try
{
    a.DoPurchase(5825, p4); //credit limit exceeded
}
catch (AccountException e) { Console.WriteLine(e.Message); }
try
{
    c.Withdraw(1500, p6); //no overdraft
}
catch (AccountException e) { Console.WriteLine(e.Message); }

try
{
    Bank.GetAccount("CK-100018"); //account does not exist
}
catch (AccountException e) { Console.WriteLine(e.Message); }

try
{
    Bank.GetPerson("Trudeau"); //user does not exist
}
catch (AccountException e) { Console.WriteLine(e.Message); }

foreach (Account account in Bank.ACCOUNTS)
{
    Console.WriteLine("\nBefore PrepareMonthlyReport()");
    Console.WriteLine(account);

    Console.WriteLine("\nAfter PrepareMonthlyReport()");
    account.PrepareMonthlyReport();    //all transactions are cleared, balance
changes
    Console.WriteLine(account);
}
```