15 Methods and Exceptions

May 12, 2022

1 Methods and Exceptions

Below is the code you've made so far. Balance is gotten by summing up the lists of transactions, but you haven't written a way to add a transaction. This happens plenty of times in coding, where to make something more robust, you have to take a step back before going forward.

Run the code cells below.

[1]:
```
public class Transaction
{
    // Properties
    public decimal Amount { get; }
    public DateTime Date { get; }
    public string Notes
    {
        get;

    }

    // Constructor
    public Transaction(decimal amount, DateTime date, string note)
    {
        this.Amount = amount;
        this.Date = date;
        this.Notes = note;
    }
}
```

[2]:
```
using System.Collections.Generic;

public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance
    {
        get
        {
            decimal balance = 0;
            foreach (var item in allTransactions)
            {
                balance += item.Amount;
            }

            return balance;
        }

    }
    private static int accountNumberSeed = 1234567890;
    private List<Transaction> allTransactions = new List<Transaction>();

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        this.Number = accountNumberSeed.ToString();
        accountNumberSeed++;

    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

[3]:
```
// Testing Code

var account = new BankAccount("Kendra", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner}␣
 ↪with {account.Balance} dollars");
```

```
Account 1234567890 was created for Kendra with 0 dollars
```

1.1 #1: Adding deposits

First things first, time to make a deposit function. This addition will make a transaction listing the amount, date, and a note that you're depositing, and then adds it to the transaction list.

Add this code inside MakeDeposit.

```
    var deposit = new Transaction(amount, date, note);
    allTransactions.Add(deposit);
```

[5]:
```
using System.Collections.Generic;

public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance
    {
        get

        {
            decimal balance = 0;
            foreach (var item in allTransactions)
            {
                balance += item.Amount;
            }

            return balance;
        }


    }
    private static int accountNumberSeed = 1234567890;
    private List<Transaction> allTransactions = new List<Transaction>();

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        this.Number = accountNumberSeed.ToString();
        accountNumberSeed++;

    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
        //Add code here!
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

1.2 #2: Exceptions

But what if someone tries to deposit negative money? That doesn't make logical sense, but currently the method allows for that. What you can do is make an exception. Before doing anything, you check that the amount deposited is more than 0. If it is, awesome, the code moves on to adding the transaction. If not, the code throws an exception, where it stops the code and prints out the issue.

Place this code in the very beginning of the `MakeDeposit` method.

```
if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of deposit must be positi
    }
```

[6]:
```
using System.Collections.Generic;

public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance
    {
        get

        {
            decimal balance = 0;
            foreach (var item in allTransactions)
            {
                balance += item.Amount;
            }

            return balance;
        }

    }
    private static int accountNumberSeed = 1234567890;
    private List<Transaction> allTransactions = new List<Transaction>();

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        this.Number = accountNumberSeed.ToString();
        accountNumberSeed++;
    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
        //Add Code here!

        var deposit = new Transaction(amount, date, note);
        allTransactions.Add(deposit);
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

1.3 #3: Adding withdrawal

Now you need to do the same thing for the withdrawal!

Add this code to MakeWithdrawal.

```
if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be pos
    }
    if (Balance - amount < 0)
    {
        throw new InvalidOperationException("Not sufficient funds for this withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
```

[7]:
```
using System.Collections.Generic;

public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance
    {
        get

        {
            decimal balance = 0;
            foreach (var item in allTransactions)
            {
                balance += item.Amount;
            }

            return balance;
        }


    }
    private static int accountNumberSeed = 1234567890;
    private List<Transaction> allTransactions = new List<Transaction>();

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        this.Number = accountNumberSeed.ToString();
        accountNumberSeed++;

    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
        if (amount <= 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Amount of␣
 ↪deposit must be positive");
        }
        var deposit = new Transaction(amount, date, note);
        allTransactions.Add(deposit);
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
        //Add code here!
    }
}
```

1.4 #4: Creating initial deposit

Now that you have deposits and withdrawals, you can finally make an initial deposit again. What you'll do is create a deposit of the initial amount when you're first constructing the bank account.

Add this code to the `BankAccount` constructor.

```
MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
```

[8]:
```
using System.Collections.Generic;

public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance
    {
        get

        {
            decimal balance = 0;
            foreach (var item in allTransactions)
            {
                balance += item.Amount;
            }

            return balance;
        }


    }
    private static int accountNumberSeed = 1234567890;
    private List<Transaction> allTransactions = new List<Transaction>();

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        this.Number = accountNumberSeed.ToString();
        accountNumberSeed++;
        //(Paste here!)

    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
        if (amount <= 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Amount of␣
 ↪deposit must be positive");
        }
        var deposit = new Transaction(amount, date, note);
        allTransactions.Add(deposit);
    }
    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
        if (amount <= 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Amount of␣
↪withdrawal must be positive");
        }
        if (Balance - amount < 0)
        {
            throw new InvalidOperationException("Not sufficient funds for this␣
↪withdrawal");
        }
        var withdrawal = new Transaction(-amount, date, note);
        allTransactions.Add(withdrawal);
        }
}
```

2 Check and test your work

There's some added line in the test code, because you can now make deposits and withdrawals. Test it out!

Run the following cells, including the new stuff in the test code.

Make your own deposit and withdrawal.

[9]:
```
public class Transaction
{
    // Properties
    public decimal Amount { get; }
    public DateTime Date { get; }
    public string Notes { get; }

    // Constructor
    public Transaction(decimal amount, DateTime date, string note)
    {
        this.Amount = amount;
        this.Date = date;
        this.Notes = note;
    }
}
```

[10]:
```
using System.Collections.Generic;

public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance
    {
        get

        {
            decimal balance = 0;
            foreach (var item in allTransactions)
            {
                balance += item.Amount;
            }

            return balance;
        }


    }
    private static int accountNumberSeed = 1234567890;
    private List<Transaction> allTransactions = new List<Transaction>();

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {

        this.Owner = name;
        this.Number = accountNumberSeed.ToString();
        accountNumberSeed++;
        MakeDeposit(initialBalance, DateTime.Now, "Initial balance"); //(#4)

    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
        //(#2)
        if (amount <= 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Amount of␣
↪deposit must be positive");
        }
        //(#1)
        var deposit = new Transaction(amount, date, note);
        allTransactions.Add(deposit);
    }
    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
        //(#3)
        if (amount <= 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Amount of␣
↪withdrawal must be positive");
        }
        if (Balance - amount < 0)
        {
            throw new InvalidOperationException("Not sufficient funds for this␣
↪withdrawal");
        }
        var withdrawal = new Transaction(-amount, date, note);
        allTransactions.Add(withdrawal);
    }
}
```

[11]:
```
var account = new BankAccount("Kendra", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner}␣
↪with {account.Balance} dollars");

account.MakeWithdrawal(500, DateTime.Now, "Rent payment");  //Added test code
Console.WriteLine(account.Balance);
account.MakeDeposit(100, DateTime.Now, "Friend paid me back");
Console.WriteLine(account.Balance);
```

```
Account 1234567890 was created for Kendra with 1000 dollars
500
600
```

3 Review

You did it! You've now successfully made a bank account class that has the following attributes:

It has a 10-digit number that uniquely identifies the bank account.

It has a string that stores the name or names of the owners.

The balance can be retrieved.

It accepts deposits.

It accepts withdrawals.

The initial balance must be positive.

Withdrawals cannot result in a negative balance.

[ ]:
```

```