|  |  |
| --- | --- |
| Ekonomiklass | 23 september 2021 |

**Beskrivning**:

I den här övningen ska vi skapa ett projekt som får utgöra grunden till en klass för ekonomiska beräkningar.

**Kursplanstermer som berörs av uppgiften**:

|  |  |
| --- | --- |
| Mål | För godkänt krävs |
| Kunskaper kring typer, variabler, operationer, uttryck, villkorssatser och loopar inom programmering. | Den studerande redogör för hur typer, variabler, uttryck, villkorssatser och loopar används inom programmering. |
| Kunskap kring namngivning och kodstruktur av klasser, metoder och variabler i objektorienterade program. | Den studerande redogör för hur namngivning och kodstruktur av klasser, metoder och variabler i objektorienterade program används. |
| Lösa problem i ett datalogiskt sammanhang. | Den studerande löser problem i datalogiska sammanhang genom att exempelvis bryta ner problemen. |
| Utveckla program med en tydligt objektorienterad struktur. | Den studerande utvecklar program med en tydligt objektorienterad struktur. |
| Förstå och använda sig av datastrukturer inom programmering. | Den studerande använder sig av datastrukturer i sin mjukvaruutveckling. |
| Planera, designa och implementera gränssnitt utifrån användaren. | Den studerande planerar, designar och implementerar gränssnitt utifrån användaren. |
| Utveckla felfria fristående program. | Den studerande skapar felfria fristående program. |

Vi låter oss inspireras av <https://www.fortnox.se/blogg/vad-ar-debet-och-kredit/>

**Termer för övningen**:

* Transaktion = överföring av pengar från ett konto till ett annat
* Debit = Vänster kolumn
* Kredit = Höger kolumn
* Kontoplan = Lista på konton
* Constructor = Första metoden som körs när en klass instansieras
* List<> = Generisk lista som hanterar vilken typ man nu ger den
* String.PadLeft = Fyller strängen med mellanslag till vänster
* String.PadRight = Fyller strängen med mellanslag till höger
* ToString() = Metod som kallas när man försöker skriva ut eller debugga en instans av en klass
* ToString(”0000”) = kollar att siffran innehåller mist 4 tecken, annars lägger den till nollor innan siffran. 532 blir 0532.

**Psuedokod:**

1. Skapa ett kontoplans konto klass
2. Skapa en transktionsklass som ska hantera debit och kredit
3. Lägga transaktioner i en lista
4. Lista alla transaktioner
5. Lista ett specifikt kontos debit och kredit
6. Kontrollräkna alla transaktioner
7. Skydda informationen
8. Kontrollera inputs

**Projektinstruktioner:**

Vi ska använda en minimal kontoplan för detta

1910 Kassa  
1930 Företagskonto  
2440 Leverantörsskuld  
3010 Försäljning  
5410 Datorinköp

**Kodning:**

Vi börjar med att skapa ett c#, .net konsolprojekt.

Skapa nu ett Github repo för ditt projekt (om du inte sparar i ett redan existerande repo)

Första klassen: Account.cs

Den ska hantera balansen mellan Debet och Kredit.

Vi börjar med att göra den osmart, den ska alltså bara ta emot data och inte ifrågasätta något.

public class Account

{

    private string label; // Benämning

    public string Label

    {

        get { return label; }

        set { label = value; }

    }

}

Skapa nu fler properties i klassen, för variablerna   
int Number // Kontonummer

double debit; // Mängd pengar

double credit; // Mängd pengar

Nu har du en klass för att kunna registrera transaktioner i konton, exempelvis.

|  |  |  |  |
| --- | --- | --- | --- |
| Kontonummer | Benämning | Debet | Kredit |
| 1910 | Företagskonto | 0 | 50000 |
| 2440 | Leverantörsskuld | 50000 | 0 |

Men det kräver två konton för att kunna göra sådana överföringar och klasser hanterar bara ett konto i taget. Alltså får vi skapa en klass som kan hantera båda konton.

Nu kan vi testa koden från Main()

static void Main()

{

Först skapar vi en lista som ska hålla koll på transaktionerna.

    List<Account> transactions = new List<Account>();

Sedan skapar vi första transaktionen.

    Account acc = new Account();

    acc.Number = 1910;

    acc.Label = "Kassa";

    acc.Credit = 5000;

Nu lägger vi till den i vår lista och så skapar vi nästa del av transaktionen

    transactions.Add(acc);

    acc = new Account();

    acc.Number = 2440;

    acc.Label = "Lev.skuld";

    acc.Debit = 5000;

…och så lägger vi den i listan

    transactions.Add(acc);

Nu kan vi skriva ut listan och verifiera att det fungerar. Vi börjar med två variabler som ska hålla koll på summan.

    double debit = 0;

    double credit = 0;

Sen kan vi loopa igenom alla transaktionerna, skriva ut dem och summera Debet och Kredit.

    foreach (var trans in transactions)

    {

        Console.Write("Konto:" + trans.Number + " " + trans.Label);

        Console.Write(" " + trans.Debit);

        Console.WriteLine(":" + trans.Credit);

        debit += trans.Debit;

        credit += trans.Credit;

    }

Slutligen skriver vi ut summan av Debet och Kredit för att kontrollera att allt gick bra

    Console.WriteLine("Summa Debet :" + debit);

    Console.WriteLine("Summa Kredit:" + credit);

    Console.WriteLine("Resultat skillnad:" + (debit-credit));

}

Underbart! Det fungerar.

Enda nackdelen nu är att det är himla omständligt att skriva in en transaktion. Det måste vi förenkla.

Vi kan skapa en metod i vår klass som hjälper till med detta, men då det kan i framtiden finnas andra klasser som ska vara med och leka med transaktionerna så behöver vi börja med att skapa en bokföringsklass.

Andra klassen: Accounting.cs

Vi börjar lite lätt med att lägga in vår lista med transaktioner där.

public class Accounting

{

    List<Account> Transactions = new List<Account>();

Men till skillnad från i mainklassen så gör vi om vår lista till en property, ersätt koden med

public class Accounting

{

    public List<Account> Transactions { get; set; } = new List<Account>();

Nu kan vem som helst som instansierar klassen nå den. Nästa steg blir att förenkla inmatandet av transaktioner.

För detta behöver vi en metod. Den ska ta emot de nödvändiga variablerna för att skapa en transaktion.

Public void CreateTransaction(int accountNr, string label, double debit, double credit)

{

    transactions.Add

        (

        new Account{

            Number = accountNr, Label = label, Debit = debit,Credit = credit

        });

}

Då går vi tillbaka till main klassen och testar.

Vi gör samma beräkningar.

static void Main()

       {

           Accounting bok = new Accounting();

**bok.CreateTransaction(1910, "Kassa", 0, 5000);**

**bok.CreateTransaction(2440, "Lev.Skuld", 5000, 0);**

           double debit = 0;

           double credit = 0;

           foreach (var trans in bok.Transactions)

           {

               Console.Write("Konto:" + trans.Number + " " + trans.Label);

               Console.Write(" " + trans.Debit);

               Console.WriteLine(":" + trans.Credit);

               debit += trans.Debit;

               credit += trans.Credit;

           }

           Console.WriteLine("Summa Debet :" + debit);

           Console.WriteLine("Summa Kredit:" + credit);

           Console.WriteLine("Resultat skillnad:" + (debit - credit));

       }

Koden blev helt klart kortare.

Det finns några saker som kan verka störande i inmatningen nu. Det ena är att vi måste skriva in benämning på kontot hela tiden, det ska räcka med numret tycker man. Ett smart program ska kunna lösa detta.

Så vi fixar det.

I början av klassen Accounting, under raden med List<> lägger vi till en rad till.

public class Accounting

{

    public List<Account> Transactions { get; set; } = new List<Account>();

**public Dictionary<int, string> ChartOfAccounts = new Dictionary<int, string>();**

Vi skapar nu en Constructor för vår klass, i den fyller vi vår kontoplan med några konton vi ska använda. Vi gör den public så att den blir lätt för andra att ändra i.

public Accounting()

{

    ChartOfAccounts.Add(1910, "Kassa");

    ChartOfAccounts.Add(1930, "Företagskonto");

    ChartOfAccounts.Add(2440, "Leverantörsskuld");

    ChartOfAccounts.Add(3010, "Försäljning");

    ChartOfAccounts.Add(5410, "Datorinköp");

}

Vi ändrar även metoden som skapar transaktionerna, så att den hämtar strängen från vår dictionary.

public void CreateTransaction(int accountNr, double debit, double credit)

 {

     Transactions.Add

         (

         new Account

         {

             Number = accountNr,

**Label = ChartOfAccounts[accountNr],**

             Debit = debit,

             Credit = credit

         }

         );

 }

Nu får vi varningar från Main, att metoden vi använde inte längre finns. Vi får ändra i main så att den inte skickar med någon benämning.

static void Main()

       {

           Accounting bok = new Accounting();

**bok.CreateTransaction(1910, 0, 5000);**

**bok.CreateTransaction(2440, 5000, 0);**

Prova att köra nu och se hur det blev. Passa på att comitta till github!

Nu ska vi snygga upp hur den skrivs ut, detta gör vi i klassen Account.cs

Detta gör vi genom att overrida metoden ToString() som finns i alla objekt. Detta på grund av att den finns i objektet Object i .net och alla klasser och typer ärver från den.

Vi snyggar upp koden i den så att den blir mer som en tabell, och så gör vi att den skriver ut alla värden i vår Account klass.

Nummer kommer att ta upp fyra tecken, namnet 20 tecken och nummerfälten tar sju tecken. Det är ingen regel, det är bara så för att jag tyckte det blev snyggt. Ändra det om du vill.

public override string ToString()

{

    return number.ToString("0000") + " | " +

           label.PadRight(20) + " | " +

           debit.ToString().PadLeft(7) + " | " +

           credit.ToString().PadLeft(7) + " | ";

}

Vi går tillbaka till main och anpassar vår kod igen

static void Main()

{

    Accounting bok = new Accounting();

    bok.CreateTransaction(1910, 0, 5000);

    bok.CreateTransaction(2440, 5000, 0);

    double debit = 0;

    double credit = 0;

    foreach (var trans in bok.Transactions)

    {

**Console.WriteLine(trans);**

        debit += trans.Debit;

        credit += trans.Credit;

    }

    Console.WriteLine();

    Console.WriteLine("Summa Debet :" + debit);

    Console.WriteLine("Summa Kredit:" + credit);

    Console.WriteLine("Resultat skillnad:" + (debit - credit));

}

Testa att köra programmet och passa på att comitta ditt projekt till Git!

**Förbättra klassen**

En ekonom skulle med all sannolikhet rysa av tanken på hur vi skriver in data, så vi måste korrigera lite. Detta får du göra 😉

1. Antingen Debet eller Kredit måste vara noll, för det är inte snyggt att skriva värden i båda fälten. Gör en kontroll för det.
2. Kontrollera att det inte skickas in minusvärden till vare sig debet eller kredit
3. Kontrollera att inte debet och kredit är noll
4. Vi har inte implementerat datum, lägg till en property i klassen
   1. Ändra i koden så att man kan skapa en transaktion med datum.
   2. Ändra också så att ToString() visar datum

**Säkerställ att data inte ändras**

En grundregel inom bokföring är att det som matats in får inte ändras. Så vi får skydda upp våra transaktioner, det som skrivits in får inte ändras efteråt.

Hur gör vi detta?

I klassen Account har vi en del properties, dessa måste skyddas upp

public double Debit

{

    get { return debit; }

    set { debit = value; }

}

Vad vi gör är att vi tar bort set delen, så att man bara kan läsa dem. Ta bort set på alla properties i klassen.

public double Debit

{

    get { return debit; }

}

Då tänker du ”Duh! Kan man inte skriva till den så kan man inte använda det”, och du har alldeles rätt. Ingen kan skriva till den längre… förutom Constructorn. Då constructorn körs bara när klassen instansieras så kan vi lägga till värden i våra variabler trots att de är skrivskyddade. De får standardvärden på det sättet. Men hur gör vi för att den ska veta vad den ska lägga i variablerna, enkelt. Vi lägger till parametrar i constructorn.

public Account(int accountNr, string label, double debit, double credit)

{

    this.number = accountNr;

    this.label = label;

    this.debit = debit;

    this.credit = credit;

}

Nu är vi tvungna att ange dessa värden när vi skapar en ny instans av klassen. Den nya constructorn orsakar problem i klassen Accounting, så vi går till den för att rätta till tabben.

I Accounting hittar vi, i metoden CreateTransaction hittar vi följande rad

Transactions.Add

        (

        new Account

        {

            Number = accountNr,

            Label = ChartOfAccounts[accountNr],

            Debit = debit,

            Credit = credit

        }

        );

Den skapar en instans av Account och fyller i informationen som ska vara i dess properties, men det kan vi inte göra längre. Så vi får ändra lite.

Transactions.Add

        (

        new Account

**(**

**accountNr,**

**ChartOfAccounts[accountNr],**

**debit,**

**credit**

**)**

        );

Nu kan man köra det igen.

**Nu ser vår main ut såhär**

static void Main()

{

    Accounting bok = new Accounting();

    bok.CreateTransaction(1910, 0, 5000);

    bok.CreateTransaction(2440, 5000, 0);

    double debit = 0;

    double credit = 0;

    foreach (var trans in bok.Transactions)

    {

        Console.WriteLine(trans);

        debit += trans.Debit;

        credit += trans.Credit;

    }

    Console.WriteLine();

    Console.WriteLine("Summa Debet :" + debit);

    Console.WriteLine("Summa Kredit:" + credit);

    Console.WriteLine("Resultat skillnad:" + (debit - credit));

}

**Extra Lullull**

Nu ska vi skapa metoder i vår Accounting klass

1. Skapa två metoder,
   1. En som returnerar summan av alla Debet
   2. En som returnerar summan av alla Kredit
2. Skapa en metod som skriver ut alla transaktioner
3. Skapa en metod som skriver ut alla transaktioner med ett specifikt konto (inparameter)
4. Då vi skriver ut alla transaktioner från vår klass, så behöver inte transaktionslistan vara publik längre, ändra den till privat – då kan ingen ändra i den utom genom klassens publika metoder.

Nu ska vi snygga vår main().

***Tips****! Du kan skapa metoderna med kod inspirerad från koden i main.*

Såhär borde den bli med de nya metoderna

static void Main()

{

    Accounting bok = new Accounting();

    bok.CreateTransaction(1910, 0, 5000);

    bok.CreateTransaction(2440, 5000, 0);

**double debit = bok.SumDebit();**

**double credit = bok.SumCredit();**

**bok.PrintAll();**

    Console.WriteLine();

    Console.WriteLine("Summa Debet :" + debit);

    Console.WriteLine("Summa Kredit:" + credit);

    Console.WriteLine("Resultat skillnad:" + (debit - credit));

}

Klart snyggare, nu ska vi snygga upp Account klassen också.

Alla properties som inte har kontrollkod, exempelvis

private string label; // Benämning

public string Label

{

    get { return label; }

}

kan ersättas med en så kallad autoproperty

public string Label { get; }

Hade den haft set delen också, hade vi skrivit { get; set; }

På så sätt kan man göra sina properties i en rad i stället för fem

.

*När du är klar med projektet, pusha allting till GitHub!*

**Sammanfattning:**

**Vad har vi lärt oss av detta exempel?**

1. Hur man använder klasser för att samla ihop information som hör ihop.
2. Hur man kontrollerar inputs innan man sparar data i listan.
3. Att ibland räcker det inte med att koden fungerar, den ska uppfylla regler som finns i den verkliga världen – som bokföringsregler.
4. Bra med research innan man börjar skriva kod.

**Vad kan göras bättre?**

1. Programmet kan göras större med fler funktioner.
2. Inmatning av flera transaktioner på en gång, så att man kan kolla slutsumman på alla innan man sparar.