Björn Persson

Visual Basic.NET för komponenter

Komponentbaserad applikationsutveckling

juni 2007

Om denna sammanfattning

Avsikten med denna sammanfattning är att sammanfatta Visual Basic.NET (VB.NET) som krävs för att utveckla komponenter samt skillnader mellan VB.NET och Visual Basic 6 (VB6). P.g.a. likheter med Java (och att Java används/har använts på komponentkurs jag undervisade) så görs även en del referenser till Java för jämförelse. Observera att vissa beskrivningar kan vara ofullständiga, bl.a. för att spara plats eller för att de inte är relevanta för en förståelse av VB.NET (som sammanfattning syftar till att beskriva). Denna sammanfattning utgår från sammanfattningen *Visual Basic 6 för komponenter*. För att få ut mest av denna sammanfattning så krävs det en grundläggande förståelse för objektorienterad programmering och databaser/SQL. Observera att detta **inte är en ersättning till eventuell kurslitteratur**.

Denna sammanfattning bygger på VB.NET (v.1.x), men mycket av stoffet bör även kunna appliceras i språk så som C# och Managed C++ (C++.NET) eller andra .NET-språk.

Första kapitlet tittar på variabler, objekt, klasser, vektorer av klassen Collection och andra detaljer (som i sammanfattningen *Visual Basic 6 för komponenter*). En introduktion till objektorienterad programmering i VB.NET tas upp i kapitel två och i tredje kapitlet behandlas felhantering (*exceptions*). Nästföljande tre kapitel fortsätter med databasprogrammering med ADO.NET. För beskrivning hur programmeringsmiljöerna VS.NET och SharpDevelop fungerar, se sammanfattningen *Använda Visual Studio.NET och SharpDevelop*.

Koden i denna sammanfattning har skrivits i Microsofts *Visual Studio.NET 1.0* (i Windows XP, d.v.s. COM+), men bör även fungera i VS.NET 2003. Koden har sedan kopierats in i dokumentet, d.v.s. fel kan ha införts av misstag vid redigering av dokumentet. Övrig programvara från Microsoft som använts är *Access XP* (samt *Visio XP* för klassdiagram).

Konventioner i sammanfattning

I texten visas metoder med parenteser efter (t.ex. **Print()**) för att visa att det är en metod. Metoder och egenskaper som tillhör ett objekt visas med fet stil och punkt före (t.ex. **.Open()** resp. **.Source**) för att visa att dom är en del av objektet som stycket eller avsnittet behandlar.

Vissa engelska begrepp saknar (enligt mig) generellt accepterade översättningar och är därför skrivna på engelska för att kunna relatera till begreppen i engelsk litteratur. I de fall då översättning används så följs det översatta ordet första gången med det engelska inom parentes. Dessa ord skrivs i kursiv stil för att visa att de har ”lånats”, t.ex. *data provider* och datakällor (*data sources*). Kursiv stil används även för hänvisningar till kapitel, avsnitt eller andra sammanfattningar.

Kod har skrivits med typsnitt av fast bredd (`Courier New`) för att göras mer lättläst samt längre exempel har inneslutits i en ram (se exempel nedan).
I Visual Basic kan programsatser (*statements*) skrivas på flera rader genom att använda understrykningstecknet (”_”). Detta underlättar läsning av kod då man slipper skrolla i sidled för att läsa längre programsatser. Viktigt är att placera ett mellanslag innan understrykningstecknet som i detta exempel:

```
enVariabel = 1 _
             + 2    'Kommentarer i kod skriv med fet stil
```

Jag är givetvis tacksam för alla konstruktiva synpunkter på sammanfattningens utformning och innehåll.

Eskilstuna, juni 2007

Björn Persson

E-post: (se startsida för min webbplats)
Personlig hemsida: `http://www.kiltedviking.net/`

Innehållsförteckning

1 Deklarera variabler och grundläggande strukturer

Visual Basic.NET (VB.NET) påminner mycket om Visual Basic 6 (VB6), dock med några skillnader. Några av dessa skillnader är att allt nu bygger på klasser (som i Java) och att vi kan skapa program utan grafiskt gränssnitt, d.v.s. textbaserade.

I detta kapitel tittar vi på skillnaderna mellan Visual Basic 6 och Visual Basic.NET (jämför med sammanfattningen *Visual Basic 6 för komponenter*).

1.1 Datatyper i Visual Basic.NET

I VB.NET (och andra .NET-språk) skiljer man på (främst) två olika sorters datatyper för variabler: värdetyper och referenstyper. En **värdetyp** är en variabel som håller sitt värde inom sitt allokerade minnesutrymme (d.v.s. på stacken) medan en **referenstyp** är en pekare till ett minnesutrymme med variabelns värde (pekare allokeras på stacken medan värdet allokeras i *heap*). Värdetyper är alltså statiska medan referenstyper är dynamiska. Ett av skälen till att skilja på dessa typer av variabler är att användandet av värdetyper kan vara mer effektivt än referenstyper (eftersom värdetyper är statiska – kan avgöras vid kompilering).

Värdetyperna motsvaras av alla numeriska datatyper (`Integer`, m.fl.), `Boolean`, `Char`, `Date`, strukturer (poster) och uppräkningstyper (*enumerations*). Övriga datatyper – d.v.s. strängar (`String`), vektorer och klasser – är referenstyper. Eftersom strängar är en datatyp som är grundläggande så har den dock många av de egenskaper som vi förväntar oss av värdetyper.

VB.NET har i stort sett samma ”enkla” (primitiva) datatyper som i VB6:

- heltal: `Byte`, `Short` (`System.Int16`), `Integer` (`System.Int32`), `Long` (`System.Int64`)
- decimaltal: `Single`, `Double`, `Decimal`
- boolesk (sant/falskt): `Boolean`
- text och strängar: `Char`, `String`
- datum: `Date`, `DateTime`

Datatyperna i listan ovan motsvarar någon datatyp i namnutrymmet `System` – datatyper som bör användas för att kod ska kunna delas mellan de olika språken i .NET. Exempelvis så motsvarar datatypen (eller snarare det reserverade ordet) `Integer` datatypen `System.Int32` – ett faktum som vi inte behöver bry oss allt för mycket om ☺. En annan viktig skillnad är att i VB6 är datatypen `Integer` två byte stor och `Long` fyra byte. Fr.o.m. VB.NET så är en `Integer` fyra byte och en `Long` åtta byte samt datatypen `Short`, med storleken två byte, har introducerats. Därmed så stämmer storleken på dessa datatyper mer överens med andra språks datatyper.

Istället för datatypen `Currency` (i VB6) använder man `Decimal` i VB.NET (med bättre precision) samt istället för `Variant` (i VB6) kan man använda `Object` i VB.NET (som är referenstyp). Observera dock att variabler av typen `Object` i VB.NET (precis som `Variant` i VB6) påverkar prestanda negativ (p.g.a. behovet av konvertering). Och istället för strukturer (d.v.s. egen definierade datatyper) kan vi använda klasser. ☺

Ytterligare en skillnad mot VB6 är att konstanter (med prefixet ”vb” i VB6) har ersatts av uppräkningstyper (*enumerations*) i VB.NET. Fördelen med uppräkningstyper är att man kan samla flera konstanter som hör ihop i en logisk enhet. Vi kommer att stöta på ett antal uppräkningstyper i denna sammanfattning.

1.1.1 Namnutrymme, referenser och importer

Ett **namnutrymme**[1] är bl.a. ett sätt att logiskt kunna dela upp klasser (och gränssnitt) samt fungerar som paket (*packages*) i Java. Genom att använda namnutrymmen undviks bl.a. namnkonflikter bland klasser, d.v.s. klasser med samma namn kan finnas i flera olika namnutrymmen. Namnutrymmen har dock inget med klasshierarkier att göra, även om namnutrymmen i sig kan vara hierarkiska. Ett av de viktigaste namnutrymmena är `System` som bl.a. innehåller alla enkla datatyper. (Se exempel nedan på fler namnutrymmen.)

För att kunna använda klasser i ett namnutrymme måste vi sätta en ”**referens**” till namnutrymmet (eller snarare *assembly*/fil, som namnutrymme finns i – se *.NET Framework* i kommande kapitel) för att kompilera. I programmeringsmiljöer så som Visual Studio.NET (VS.NET) och SharpDevelop kan man referera till ett namnutrymme genom att högerklicka på noden References i Solution Explorer respektive Projects-fönstret och bläddra sig fram till namnutrymmet. När vi skapar ett nytt projekt i Visual Studio.NET så finns redan några referenser till de vanligaste namnutrymmena, bl.a. `System`. (Om kompilering sker i kommandotolken används istället en växel till kompilatorn för att ange referenserna.)

När vi satt en referens till ett namnutrymme så kan vi använda klasser i namnutrymmet – dock med deras fullständiga namn, d.v.s. både namnutrymme och klass (t.ex. `System.Convert`). För att slippa upprepa namnutrymmen kan vi **importera** namnutrymmet, vilket vi gör med direktivet `Imports`[2]. Alla importer måste ske längst upp i källkodsfilen som vi vill importera namnutrymmet i. Som standard så är alltid namnutrymmet `System` importerat.

Nedan visas hur man använder direktivet `Imports`:

```
Imports System                'Överflödigt då System alltid är importerat
Imports System.Data.OleDb     'Import av OLEDB-klasser i ADO.NET
```

Referenser visar på beroenden av andra filer (*assemblies*) medan importer är till för att slippa behöva skriva fullständiga namn på klasser.

1.1.2 Enkla (primitiva) datatyper och strängar i VB.NET

I VB.NET har vi ett antal enkla (eller primitiva) datatyper, d.v.s. datatyper som vi använder för att bl.a. bygga andra datatyper (strukturer och klasser). Enkla datatyper är ofta de datatyper som vi förutsätter finns i både strukturerade och objektorienterade språk. I VB.NET så är dessa främst värdetyper – ett undantag är dock strängar som är en referenstyp (av klassen `String`). En viktig egenskap hos enkla datatyper är att när vi tilldelar ett värde från en variabel till en annan så kommer den andra variabeln att innehålla en kopia av originalets värde.[3] Detta är något som är viktigt att tänka på när vi använder främst strängar.

Strängar (av typen `String`) går inte att ”ändra” (eller snarare objekten som strängar faktiskt är). Om vi vill ändra strängars längd, t.ex. genom att sammanfoga med en annan sträng, så kommer helt nytt minne att allokeras för de nya strängarna. Minnet för de ursprungliga strängarna kommer att deallokeras först när *garbage collection* ”slår till” (vilket kan bli ett problem i loopar där vi bygger strängar för t.ex. utskrift – se exempel nedan).

---

[1] Ett namnutrymme motsvarar ett paket (*package*) i Java.
[2] I C# används det reserverade ordet `using` istället.
[3] Om samma tilldelning skett med referenstyper så hade den andra variabeln refererat till samma objekt som första variabeln (se avsnitt om objekt nedan). Eftersom klassen `String` är en enkel (primitiv) datatyp så fungerar den som värdetyper vid bl.a. tilldelning trots att det är en referenstyp!

```
Dim strTemp As String
strTemp = "Hej"                       'Här allokeras minne för strTemp
strTemp = strTemp & " hela världen"  'Här allokeras nytt minne för (hela) strTemp

Dim i As Integer
For i = 0 To 1000
  strTemp = strTemp & CStr(i)         'Minne kommer allokeras 1001 gånger!!
Next
```

Om vi har strängar som ändras så bör vi (istället för datatypen `String`) använda klassen `StringBuilder` för att effektivisera manipulation av strängar. Detta är en klass som vi måste skapa en instans (objekt) av och istället för &-operatorn använder vi metoden `Append()` för att lägga till text sist i strängen.

```
Dim strTemp As New System.Text.StringBuilder()
strTemp = "Hej"                       'Här allokeras minne för strTemp
strTemp.Append(" hela världen")     'Här allokeras nytt minne för sträng i argument

Dim i As Integer
strTemp = New System.Text.StringBuilder()

For i = 0 To 1000
  strTemp.Append(" ")
  strTemp.Append(CStr(i))
Next
```

1.1.3 Strukturer

Om vi vill använda strukturer så använder vi det reserverade ordet `Structure` för att definiera en struktur. Varje medlem i en struktur måste deklareras som `Public`, `Private`, m.m.. En struktur fungerar på många sätt som en klass, men kan bl.a. inte ärva eller ärvas från. En viktig skillnad är dock att variabler av en strukturtyp är värdetyper (medan variabler av klasser är referenstyper), d.v.s. instanser behöver inte skapas. I nedanstående exempel definieras en struktur Person samt en variabel av typen deklareras och tilldelas värden.

```
Public Structure Person             'Definiera struktur
  Public Namn As String
  Public Adress As String
End Structure

Dim Jag As Person                   'Deklarera variabel av struktur
Jag.Namn = "Björn"                  'Tilldela värden till strukturs medlemmar
Jag.Adress = "Eskilstuna"
```

1.1.4 Deklarera variabler

Deklarationer av variabler sker på liknande sätt i VB.NET som i VB6, d.v.s. med de reserverade orden `Dim`, `Public`, `Protected` eller `Private`. En viktig skillnad är att vi numera kan tilldela initiala värden när vi deklarerar en variabel samt att vi kan deklarera flera variabler av samma typ utan att behöva upprepa datatypen.

```
Dim intA As Integer = 0           'Deklarera och tilldela initialt värde samtidigt
Dim intB, intC, intD As Integer  'Deklarera flera variabler av samma typ
Dim strA As String, intE, intF As Integer  '... och variabler av olika typer
```

Observera också att vektorer (*arrays*) har blivit objekt, ”nollbaserade” (index börjar på 0) och att vi deklarerar dem med högsta värdet på index! D.v.s. när vi deklarerar en vektor av storleken N så använder vi värdet N-1 vid deklaration/skapande.

```
Dim  arrDagar(6) As Integer                'Vektor av storleken 7 (index 0 till 6)
Dim  arrPos() As String = New String(2) {} 'Vektor av storleken 3
```

1.1.5 Tilldela variabler

Tilldelning av värden till variabler i VB.NET sker på samma sätt som i VB6 (se även *Objekt i Visual Basic.NET* nedan), d.v.s. med likhetstecken (=). Observera dock att likhetsjämförelse i VB.NET också sker med likhetstecken.

1.1.6 Konvertering av värden

Även om VB.NET (och VB6 ☺) tillåter implicit konvertering av värden så är det oftast bättre (och mer effektivt) att använda explicit konvertering. I VB.NET finns det lite olika sätt att konvertera explicit.

- CXxx-funktioner – konverterar en sträng (uttryck) eller tal till typ ”Xxx”.
- Klassmetoder[4] i `System.Convert` – konverterar från en typ till en annan.
- CType-funktionen – konverterar ett uttryck till en viss typ.

Vilket sätt vi använder bör avgöras med tanke på effektiviteten hos konverteringen och vad vi vill konvertera.

1.1.6.1 CXxx-funktionerna

För varje enkel (primitiv) datatyp i VB.NET (och bara i VB.NET!) finns en konverterings-funktion som tar en sträng (uttryck) eller tal som parameter och returnerar ett värde av motsvarande datatyp. Dessa funktioner är i stor sett samma som finns i VB6.

- `CBool()` – returnerar en `Boolean`
- `CByte()` – returnerar en `Byte`
- `CChar()` – returnerar en `Char`
- `CDate()` – returnerar en `Date`
- `CDbl()` – returnerar en `Double`
- `CDec()` – returnerar en `Decimal`
- `CInt()` – returnerar en `Integer`
- `CLng()` – returnerar en `Long`
- `CShort()` – returnerar en `Short`
- `CSng()` – returnerar en `Single`
- `CStr()` – returnerar en `String`

```
Dim strText As String = "42"
Dim intTemp As Integer, datIdag As Date

intTemp = CInt(strText)
datIdag = CDate("2002-12-12")
```

Observera även att fel genereras om funktionerna ovan inte kan konvertera uttryck (t.ex. `CInt("42år")`). En annan nackdel med dessa funktioner är att de inte är de mest effektiva… Klassen `Convert` nedan har överbelastade[5] metoder som är mer effektiva.

[4] Klassmetoder deklareras som `Shared` i VB.NET (och `static` i C#).

1.1.6.2 System.Convert

I namnutrymmet System finns en klass `Convert` med ett antal överbelastade klassmetoder[6], d.v.s. typen på parametern som skickas till metoden avgör vilken ”version” av metoden som anropas.

Exempel på några metoder i klassen visas nedan (sök på ” System.Convert class” i hjälpen för en komplett lista). Observera att här är ett tillfälle då vi behöver veta att datatypen Integer motsvaras av datatypen System.Int32.

- `ToBoolean()`
- `ToByte()`
- `ToChar()`
- `ToDateTime()`
- `ToDecimal()`
- `ToDouble()`
- `ToInt16()` – returnerar en `Short`
- `ToInt32()` – returnerar en `Integer`
- `ToInt64()` – returnerar en `Long`
- `ToSingle()`
- `ToString()`

```
Dim strText As String = "42"
Dim intTemp As Integer

intTemp = Convert.ToInt32(strText)
```

1.1.6.3 CType-funktionen

Funktionen `CType()` i VB.NET har två parametrar – den första konverteras till den typ som skickas i den andra. Observera att den andra parametern ska vara typen och inte en sträng innehållande typen.

```
Dim strText As String = "42"
Dim intTemp As Integer

intTemp = CType(strText, Integer)  'Konvertera värde i strText till Integer
```

Denna funktion bör även användas när vi vill konvertera ett objekt från förälderklass till barnklass (*downcasting* – se mer nästa avsnitt).

1.2 Objekt i Visual Basic.NET

Variabler som refererar till (pekar på) objekt är referenstyper. I detta avsnitt beskrivs lite grundläggande saker som har med referenstypers likhet med värdetyper, syntax, m.m. samt i nästa kapitel beskrivs hur klasser och gränssnitt skapas, arv fungerar, m.m..

1.2.1 Deklarera och tilldela variabler som refererar till objekt

I VB.NET behandlas objekt på samma sätt som enkla datatyper, d.v.s. vi behöver inte längre använda det (i VB6) reserverade ordet `Set` för att tilldela ett objekt till en variabel.[7] Vi måste dock fortfarande skapa objekten (instanser av klasser) med det reserverade ordet `New`.

---

[5] Överbelastade metoder är metoder med samma namn men olika typer på och/eller antal parametrar.

[6] En klassmetod är en ”statisk” metod, d.v.s. en metod som deklarerats som `Shared` i VB.NET (och `static` i C#). För att anropa dessa metoder behöver vi inte skapa en instans (objekt) av klassen utan vi skickar metoden till klassen.

1.2.2 Konvertering av objekt

Även med objekt kan vi använda funktionen `CType()` för konvertering (övriga metoder i förra avsnittet är främst avsedda för att konvertera till någon av de enkla datatyperna). I exempel nedan deklareras en variabel av typen Person och skapas en instans av subklassen Student. Sedan konverteras typen (för referensen till objektet) till klassen Student och tilldelas till en variabel av typen Student.

```
Dim objP As Person = New Student()  'Variabel av superklass – instans av subklass
Dim objS As Student

objS = CType(objP, Student)         '"Konvertera" från superklass till subklass
```

Ovanstående exempel visar på s.k. *downcasting*, konvertering av referens från superklass till subklass. Exempel ovan fungerar också utan explicit konvertering med funktionen `CType()` i VB.NET. **Innan** vi utför konverteringen bör vi kontrollera att objektet är en instans av klass vi vill konvertera till. I VB.NET använder vi de reserverade orden `TypeOf…Is`.

```
Dim objP As Person = New Student()
Dim objS As Student

  'Om objekt är av typen Student...
If TypeOf objP Is Student Then
  objS = CType(objP, Student)  '... konvertera till Student
End If
```

1.2.3 Boxing och unboxing

Boxing innebär att en variabel av värdetyp konverteras till en referenstyp (d.v.s. en instans av en klass). I exempel nedan konverteras en variabel av typen `Integer` till en variabel (d.v.s. referensen) av klassen `Object` – faktisk klass för objektet som konverteringen resulterar i är `System.Int32`. Boxing sker bl.a. när vi använder metoder som kräver ett objekt men vi skickar en värdetyp, t.ex. då vi använder någon av Collection-klasserna nedan.

```
Dim intTal As Integer = 6
Dim objTal As Object

objTal = intTal  'Boxing sker – konvertering från heltal till instans av Integer
```

Unboxing är det motsatta, d.v.s. en referenstyp som konverteras till en värdetyp.

```
Dim objTal2 As Object = 6
Dim intTal2 As Integer

intTal2 = objTal2  'Unboxing sker – konvertering från instans av Integer t heltal
```

Vi bör vi använda någon av metoderna i klassen `Convert` (t.ex. `Convert.ToInt32()` som i nedan) för konverteringen, även om *boxing* och *unboxing* kan ske implicit i VB.NET (som i två exempel ovan).

---

[7] Om ni av ”misstag” skriver Set framför en tilldelning (av ett objekt till en variabel) så kommer VS.NET att ta bort ordet.

```
Dim objTal2 As Object = 6
Dim intTal2 As Integer

intTal2 = Convert.ToInt32(objTal2)
```

1.3 Collections

Klassen Collection finns även i .NET, plus ett antal typer av vektorer till (se ”System.Collections namespace” i dokumentationen), och fungerar på liknande sätt som i VB6. Precis som i VB6 så måste vi skapa en ny instans av vektorerna.

1.4 Skapa textbaserade applikationer

En nyhet i VB.NET är att vi numera kan skapa textbaserade applikationer (eller konsol-applikationer – *Console Application*). Till skillnad mot Windows-applikationer (*Windows Application*) så skapas en modul istället för en klass i VB.NET.

I exemplen nedan skrivs först strängen ”Hello World” ut följt av en tom rad och ledtext (d.v.s ytterligare en sträng ☺) som uppmanar användaren att trycka på ENTER för att avsluta programmet. Alla utskrifter till kommandotolken (konsolen) sker med `Console.WriteLine()`. Sist i programmet läses en rad in m.h.a. `Console.ReadLine()`.

```
Module Module1

  Sub Main()
    Console.WriteLine("Hello World")     'Skriv ut lite text...
    Console.WriteLine()                  'Skriv ut en tom rad

    'Skriv ut "ledtext"
    Console.WriteLine("Tryck ENTER för att fortsätta...")
    Console.ReadLine()                   'Läs in rad (d.v.s tills ENTER)
  End Sub

End Module
```

I felsökningssyfte (*debugging*) kan vi använda `Debug.PrintLine()` för att skriva till Debug-fönstret (eller Output-fönstret). (Detta i motsats till `Debug.Print()` i VB6.) Detta kan användas för felsökning även i Windows-applikationsprojekt. (Se sammanfattningen *Använda Visual Studio.NET och SharpDevelop* för mer information om avlusning i VS.NET.)

1.5 De nya meddelanderutorna

I VB.NET kan vi fortfarande använda funktionen `MsgBox()` för att visa en meddelanderuta. Men eftersom VB.NET blivit mer objektorienterat så finns nu även klasser för att skapa meddelanderutor. En fördel med att använda klassvarianten är att det fungerar i både VB.NET och andra .NET-språk.

För att skapa en meddelanderuta använder vi (statiska) klassmetoden `Show()` i klassen MessageBox. Metoden är överbelastad, d.v.s. det finns flera versioner av den. I sin enklaste form så behöver vi bara använda parametern för meddelandet som ska visas (den första). Andra parametrar är för rubrik på meddelanderuta, vilka knappar som ska visas, ikon som ska visas samt vilken knapp som ska vara standard (*default*).

I exempel nedan visas användandet av ovan nämnda parametrar samt resultatet av metodanropet i figur övan.

```
MessageBox.Show("Meddelande", "Rubrik", MessageBoxButtons.OK, _
                MessageBoxIcon.Information, MessageBoxDefaultButton.Button1)
```

2 Objektorienterad programmering med Visual Basic.NET

VB.NET är nu ett fulländat objektorienterat språk, speciellt nu som VB.NET även stödjer arv av implementation (d.v.s. kod). I detta kapitel titta vi på hur man skapar och använder klasser i VB.NET samt en del skillnader mot klasser i VB6.

2.1 Klasser och objekt i Visual Basic.NET

Termer för objektorientering i VB.NET stämmer fortfarande inte riktigt överens med modelleringstekniker (t.ex. UML). Vi har fortfarande **egenskaper** (*properties*), **metoder** (*methods*) och **händelser** (*events*).

Tyvärr verkar det som om Class Builder från VB6 har försvunnit. (Antagligen vill Microsoft att vi köper verktyg som Visio istället.)

2.2 Klasser, moduler och formulär

I VB.NET har skillnaderna mellan klasser, moduler och formulär ”suddats ut” – allt är nu klasser (nästan). Vi kan dock fortfarande använda moduler om vi vill, vi använder då det reserverade ordet `Module` istället för `Class`. Skillnaden mellan en modul och en klass är fortfarande att vi kan skapa instanser av klasser men inte moduler. Eventuella publika variabler i en modul är globala för projektet, precis som i VB6.

En stor skillnad mellan VB6 och VB.NET är att en fil (.CLS i VB6) inte tvunget motsvarar en klass längre – vi använder de reserverade orden `Class` och `End Class` för att avgränsa klassen (d.v.s. var definitionen av klassen börjar och slutar). Vi kan därför skapa flera klasser i samma fil, om vi så önskar. (Men om vi skapar en klass i varje fil så kan vi lättare återanvända våra klasser i andra projekt.)

```
Public Class MinKlass
  'Definition av klass i VB.NET
End Class
```

2.2.1 Namn på filer i projekt

Alla ”källkodfiler” i VS.NET får numera en filändelse som visar vilket språk som koden i filen är skrivet med, t.ex. .VB för Visual Basic.NET och .CS för C#. Borta är alltså våra filer med ändelser som .FRM, .MOD och .CLS. Ett formulär har numera blivit en klass t.ex..

2.3 Egenskaper i klasser

Implementationen av egenskaper i VB.NET har ändrats något sedan VB6, men sättet att använda egenskaperna är det samma.[8] Accessmetoderna kallas numera för *setters* och *getters*.

I exemplet nedan använder vi en privat instansvariabel (`mstrNamn`[9]) för egenskapen Namn och skapar accessmetoder för att sätta och hämta värdet för egenskapen.

[8] Vi kan fortfarande deklarera en instansvariabel (egenskap) som publik, men detta vore (fortfarande) att strunta i inkapsling som (fortfarande) är halva meningen med objektorientering.

[9] Bokstaven `m` används ofta som prefix för att tala om att det är en medlem av en klass och prefixet `str` används för att tala om att det är en sträng. Dessa prefix är inte nödvändiga, men kan göra det lättare att förstå koden och vilken typ av variabel som används i klassens metoder.

```
Public Class Person
  Private mstrNamn As String      'Deklarera en privat instansvariabel f. egenskap

  Property Namn() As String       'Skapa publika accessmetoder för egenskap
    Get                           'Getter-metod
      Return mstrNamn
    End Get
    Set(ByVal Value As String)    'Setter-metod
      mstrNamn = Value
    End Set
  End Property

  ...  'Resten av klassen Person
End Class
```

Egenskaper kan även vara endast läsbara (*read only*) eller skrivbara (*write only*). Detta kan vara användbart för egenskaper som t.ex. beräknas, så som exempel med ålder nedan, eller om vi använder lösenord (som inte bör läsas). För detta används de reserverade orden `ReadOnly` respektive `WriteOnly`.

Ålder är en egenskap som inte bör lagras – den bör beräknas utifrån aktuellt år och persons födelseår. Och eftersom egenskapen inte bör lagras så bör den endast vara läsbar. Nedan lagras födelseår i en instansvariabel (`mstrFodelsear`, som det bör finnas accessmetoder för) och egenskapen Ålder beräknas utifrån systemets aktuella år och instansvariabeln.

```
Public Class Person
  Private mstrNamn As String
  Private mintFodelsear As Integer = 0

  ...  'Resten av klassen Person

  Public ReadOnly Property Alder() As Integer 'Enbart läsbar egenskap (endast Get)
    Get
      Return (DateTime.Now.Year - mintFodelsear)
    End Get
  End Property

End Class
```

2.4 Metoder i klasser

Bortsett från accessmetodernas implementation så är metoder i klasser i stort sett som i VB6. En viktig skillnad i VB.NET, och som gäller även för ”vanliga” procedurer och funktioner, är att parametrar till metoder numera skickas som värden (`ByVal`) som standard (och inte som referenser, `ByRef`). Ytterligare en skillnad är att funktioner numera kan returnera sitt värde med det reserverade ordet `Return` (istället för att tilldela värdet till funktionens namn). Exekveringen av funktionen avbryts också då `Return` påträffas, d.v.s. eventuella programsatser efter `Return` kommer inte att exekveras. I nedanstående exempelfunktion, `alder()` så hämtas året för dagens datum (`Today.Year`), som sen persons födelseår (instansvariabeln `mintFodelsear`) subtraheras från.

```
Private Function alder() As Integer
    Return Today.Year - mintFodelsear
End Function
```

2.4.1 Konstruktor och destruktor

Procedurerna `Class_Initialize()` och `Class_Terminate()` från VB6 har ersatts med procedurerna `New()` respektive `Finalize()` i VB.NET, d.v.s. konstruktor och destruktor.

En av nyheterna i VB.NET är att konstruktorn kan ta emot parametrar, och därmed kan överbelastas. Observera dock att om en konstruktor med parametrar skapas så kan inte standardkonstruktorn (d.v.s. konstruktorn utan parametrar) anropas längre om vi inte skapar även den (som i Java).

```
Public Class Bil
  Private mstrMarke As String      'Instansvariabler
  Private mintArsmodell As Integer

  Public Sub New()                 'Standardkonstruktor (utan parametrar)

  End Sub

    'Konstruktor med parametrar
  Public Sub New(ByVal strMarke As String, ByVal intArsmodell As Integer)
    mstrMarke = strMarke
    mintArsmodell = intArsmodell
  End Sub

End Class
```

I exemplet ovan tilldelas värden från konstruktorns parametrar direkt till instansvariablerna, men bättre är att använda accessmetoder (om sådan finns, vilket det bör ☺).[10] Här används accessmetoder istället – användandet av det reserverade ordet `Me` är dock inte nödvändigt. `Me` refererar till aktuell instans som metod exekverar i (som `this` i t.ex. C# och Java).

```
Public Class Bil
  Private mstrMarke As String
  Private mintArsmodell As Integer

    '*** Konstruktorer ***
  Public Sub New()

  End Sub

  Public Sub New(ByVal strMarke As String, ByVal intArsmodell As Integer)
    Me.Marke = strMarke            'Använd accessmetoder för tilldelning
    Me.Arsmodell = intArsmodell
  End Sub

    '*** Accessmetoder ***
  Property Marke() As String
    Get
      Return mstrMarke
    End Get
    Set(ByVal Value As String)
      mstrMarke = Value
    End Set
  End Property

  Property Arsmodell() As Integer
    Get
      Return mintArsmodell
    End Get
    Set(ByVal Value As Integer)
      mintArsmodell = Value
    End Set
  End Property

End Class
```

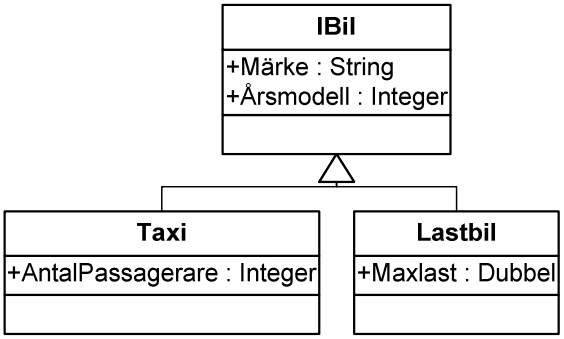
[10] Accessmetoder kan t.ex. kontrollera att värden som ska sättas är giltiga (t.ex. positiva tal för ålder).

Observera att vi **inte** har någon kontroll över **när** proceduren `Finalize()` anropas. Detta då det är upp till .NET att förstör objekt som inte längre refereras till (d.v.s. dess *garbage collection*).

2.5 Använda Class Builder för att skapa klass

Verktyget Class Builder har ”utgått” ur VS.NET – använd Microsoft Visio eller Rational Rose istället.

2.6 Arv av gränssnitt



I VB.NET kan vi numera skapa även gränssnitt (*interface*) som klasser kan implementera. En (sub-)klass kan endast ärva från en superklass men kan implementera flera gränssnitt. (I detta exempel ska vi dock endast implementera ett gränssnitt. ☺)

Precis som i VB6-exemplet[11] har alla bilar egenskaperna `Märke` och `Årsmodell`, vilket kan samlas i gränssnittet IBil. I VB.NET kan vi definiera även egenskaper i gränssnitt, men inte implementera (se även *Abstrakta klasser och arv* nedan). Klasserna Taxi och Lastbil kan sen implementera gränssnittet IBil samt utöka med egenskaperna `AntalPassagerare` respektive `Maxlast`. Vi får alltså följande modell (se bild ovan):

För att skapa klasser i VS.NET skapar vi ett Visual Basic-projekt.

1. Välj File→New…→Project för att visa dialogrutan New Project.
2. Markera **Visual Basic Projects** i listrutan (trädet) **Project Types** och sen **Windows Application** i listrutan **Templates**. Bläddra till mapp som projektets mapp ska sparas i och namnge sen projektet (t.ex. ArvBilNET) i textrutan Name.[12]
3. Skapa sen ytterligare tre filer (för gränssnittet och de två klasserna) genom att välja **Add Class…** från Project-menyn (tre gånger), ändra respektive filnamn till samma som gränssnittets/klassens namn (`IBil.vb`, `Taxi.vb` resp. `Lastbil.vb`) och klicka sen OK i dialogrutan som visas. (Vill du inte ha tre filer så räcker det med att du skapar en fil samt lägger till gränssnittet och klasserna i samma fil.)
4. Eftersom IBil är ett gränssnitt och inte en klass så måste vi byta ut det reserverade ordet `Class` mot `Interface` i filen `IBil.vb`.
5. Fyll sen i koden i styckena nedan.

(Ni behöver inte ta bort formuläret, eller snarare klassen, `Form1` då vi kommer att använda det när vi ska testa klasserna.)

2.6.1 Gränssnittet IBil

Detta gränssnitt är inga konstigheter med och koden blir följande i filen `IBil.vb`.

```
Public Interface IBil
```

[11] Se sammanfattningen Visual Basic 6 för komponenter.
[12] Genom att bläddra till en mapp (att spara projektmappen i) först så kommer VS.NET automatiskt att skapa en mapp med samma namn som projektets.

```
    Property Marke() As String
    Property Arsmodell() As Integer
  End Interface
```

2.6.2 Klassen Taxi

Att ärva gränssnitt i VB.NET är betydligt mycket enklare än i VB6[13] – vi talar om vilket gränssnitt vi vill implementera och implementerar sen alla egenskap i gränssnittet.

Steg 1: Vi anger att klassen Taxi implementerar gränssnittet IBil med det reserverade ordet `Implements`, vilket skrivs längst upp i klassen.

```
  Public Class Taxi
    Implements IBil   'Ange gränssnitt att implementera
```

Steg 2: Vi deklarerar variabler för det ärvda gränssnittet och den egna klassens egenskaper.

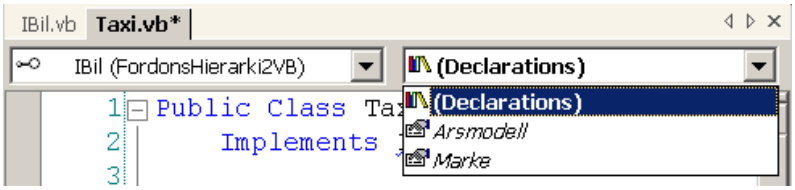
```
  Private mstrMarke As String         'Variabel för att hålla reda på IBils egenskap
  Private mintArsmodell As Integer    'Variabel för att hålla reda på IBils egenskap
  Private mintAntalPassagerare As Integer  'Var. för att hålla reda på egen egenskap
```

Steg 3: Och sist implementerar vi egenskaperna – ärvda och egen. När det gäller gränssnittets accessmetoder så måste vi tala om att vi implementerar en viss egenskap i gränssnittet. Detta gör vi genom att lägga till `Implements Gränssnitt.Egenskap` efter egenskapens datatyp.

```
    'Implementation av gränssnittet IBil
    Public Property Marke() As String Implements IBil.Marke
      Get
        Return mstrMarke
      End Get
      Set(ByVal Value As String)
        mstrMarke = Value
      End Set
    End Property

    Property Arsmodell() As Integer Implements IBil.Arsmodell
      Get
        Return mintArsmodell
      End Get
      Set(ByVal Value As Integer)
        mintArsmodell = Value
      End Set
    End Property

      'Implementation av egna gränssnittet (Taxi)
    Property AntalPassagerare() As Integer
      Get
        Return mintAntalPassagerare
      End Get
      Set(ByVal Value As Integer)
        mintAntalPassagerare = Value
      End Set
    End Property

  End Class  'Taxi
```

Signaturer för ärvda egenskaper kan VS.NET skapa åt oss genom att vi först väljer gränssnittet som implementeras (IBil) i listrutan Class Name (till vänster i bild nedan) och sen egenskapen i listrutan Method Name (till höger). Genom att göra på detta sätt så behöver vi bara ”implementera” accessmetoderna, d.v.s. fylla i koden i respektive metod.

[13] I VB6 var vi tvungen att skapa en klass med metoder utan implementation för vårt ”gränssnitt”.



Figur 1 - Infoga signaturer från ärvda egenskaper i VB.NET.

Givetvis fungerar det på samma sätt med metoder som ärvs.

2.6.2.1 Fullständig kod för klassen Taxi

```
Public Class Taxi
  Implements IBil   'Ange gränssnitt att implementera

    'Deklarera variabler för egenskaperna
  Private mstrMarke As String
  Private mintArsmodell As Integer
  Private mintAntalPassagerare As Integer

  '*****************************************************************************
  'Implementera gränssnittet IBil
  '*****************************************************************************
  Public Property Marke() As String Implements IBil.Marke
    Get
      Return mstrMarke
    End Get
    Set(ByVal Value As String)
      mstrMarke = Value
    End Set
  End Property

  Property Arsmodell() As Integer Implements IBil.Arsmodell
    Get
      Return mintArsmodell
    End Get
    Set(ByVal Value As Integer)
      mintArsmodell = Value
    End Set
  End Property

  '*****************************************************************************
  'Implementera eget gränssnitt (Taxi)
  '*****************************************************************************
  Property AntalPassagerare() As Integer
    Get
      Return mintAntalPassagerare
    End Get
    Set(ByVal Value As Integer)
      mintAntalPassagerare = Value
    End Set
  End Property

End Class
```

2.6.3 Klassen Lastbil

Klassen Lastbil ser likadan ut som klassen Taxi, med skillnaden att vi implementerar egenskapen Maxlast istället. Kod som skiljer sig från klassen Taxis visas nedan:

```
...
    'Deklarera variabler för egenskaperna
Private mdblMaxlast As Double
...
Property Maxlast() As Double
```

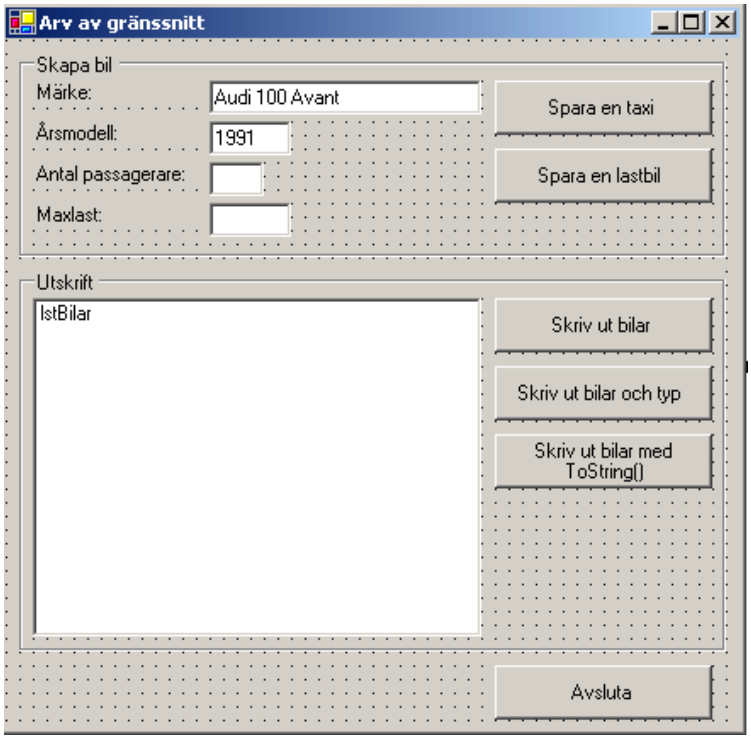
```
    Get
      Return mdblMaxlast
    End Get
    Set(ByVal Value As Double)
      mdblMaxlast = Value
    End Set
  End Property
  ...
```

2.6.4 Test av klasserna Taxi och Lastbil

2.6.4.1 Skapa formuläret

För att testa klasserna Taxi och Lastbil placerar vi följande kontroller på formuläret (vars klass vi döper om till `frmMain`):

- 4 st. etiketter (se bild)
- 4 st. textrutor (`txtMarke`, `txtArsmodell`, `txtAntalPassagerare` och `txtMaxlast`)
- 1 st. listruta med namnet `lstBilar`.
- 6 st. kommandoknappar med namnen `btnSparaTaxi`, `btnSparaLastbil`, `btnSkrivUt`, `btnSkrivUtTyp`, `btnToString` resp. `btnAvsluta`.



Figur 2 - Formuläret frmMain för att test av klasser.

Nästa steg är att deklarera en vektorvariabel (av typen Collection) som vi ska lagra bilarna vi skapar i.[14] Vi deklarerar variabeln efter Inherits-satsen i början på klassen för vårt formulär, d.v.s. som en instansvariabel för vår formulärklass.

```
  Public Class frmMain
```

[14] Vi skulle också (i.o.m. VB.NET) kunna lagra objekten direkt i listrutan då vi numera kan lagra objekt och inte bara strängar i en listruta. Men jag har valt att göra exempel som i sammanfattningen VB6 för komponenter för jämförelse.

```
    Inherits System.Windows.Forms.Form

    Dim colBilar As Collection  'Vektor att spara bilar i
```

Lättast är sen att dubbelklicka på nedanstående kontroller i formuläret (och i följande ordning) för att generera skalen till procedurerna som ska svara på olika händelser (musklickningar etc.). Nedan beskriv vad som respektive händelsehanterare gör.

1. formulärets bakgrund → `frmMain_Load()` – skapar vektorn för bilarna.
2. knappen Spara en taxi → `btnSparaTaxi_Click()` – skapar en instans av klassen Taxi, sätter egenskaper för taxin och lägger till i vektorn med bilar.
3. knappen Spara en lastbil → `btnSparaLastbil_Click()` – skapar en instans av klassen Lastbil, sätter egenskaper för lastbilen och lägger till i vektorn med bilar.
4. knappen Skriv ut bilar → `btnSkrivUt_Click()` – skriver ut bilarnas egenskaper i gränssnittet IBil till listrutan.
5. knappen Avsluta → `btnAvsluta_Click()` – avslutar programmet.

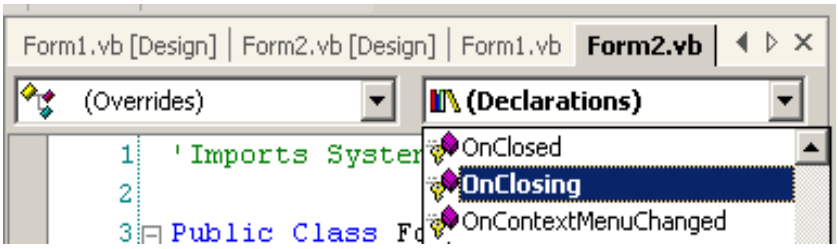
Fyll sedan i koden nedan som saknas i formulärets kodfönster.

```
  Private Sub frmMain_Load(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles MyBase.Load
    colBilar = New Collection() 'Skapa Collection-objekt
  End Sub

  '*************************************************************************

  Private Sub btnSparaTaxi_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles btnSparaTaxi.Click
    Dim objTaxi As Taxi

    Try
      objTaxi = New Taxi()           'Skapa taxiobjekt
      objTaxi.Marke = txtMarke.Text  'Ange värden för egenskaper
      objTaxi.Arsmodell = Convert.ToInt32(txtArsmodell.Text)
      objTaxi.AntalPassagerare = Convert.ToInt32(txtAntalPassagerare.Text)
      colBilar.Add(objTaxi)          'Lägg till taxi i vektor med bilar
    Catch err As Exception
      MessageBox.Show(err.ToString)
    End Try
  End Sub

  '*************************************************************************

  Private Sub btnSparaLastbil_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles btnSparaLastbil.Click
    Dim objLastbil As Lastbil

    objLastbil = New Lastbil()       'Skapa lastbilsobjekt
    objLastbil.Marke = txtMarke.Text 'Ange värden för egenskaper
    objLastbil.Arsmodell = Convert.ToInt32(txtArsmodell.Text)
    objLastbil.Maxlast = Convert.ToDouble(txtMaxlast.Text)

    colBilar.Add(objLastbil)         'Lägg till lastbil i vektor med bilar
  End Sub

  '*************************************************************************

  Private Sub btnSkrivUt_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles btnSkrivUt.Click
    Dim objbil As IBil               'Deklarear variabel av gränssnitts typ

    For Each objbil In colBilar      'Loopa över vektor och skriv ut i listruta
```

```
      lstBilar.Items.Add(objbil.Marke & ", " & objbil.Arsmodell)
    Next

    lstBilar.Items.Add(" ")          'Skriv ut tom rad i listruta
  End Sub

  '*****************************************************************************

  Private Sub btnAvsluta_Click(ByVal sender As System.Object, _
          ByVal e As System.EventArgs) Handles btnAvsluta.Click
    Me.Close()      'Stäng formulär
  End Sub

  '*****************************************************************************
  'Procedur som anropas då formulär håller på att stängas
  Private Sub Form_Closing(ByVal sender As Object, ByVal e As CancelEventArgs) _
          Handles MyBase.Closing
    Dim intAvsluta As Integer

    'Fråga användare om avsluta program
    intAvsluta = MessageBox.Show("Avsluta program?", "Avsluta", _
      MessageBoxButtons.YesNo, MessageBoxIcon.Question, _
      MessageBoxDefaultButton.Button1)

    'Om nej - avbryt avslutning av program
    If intAvsluta = DialogResult.No Then
      e.Cancel = True          'Avbryt händelse (avslutning)
    End If
  End Sub
```

Sista proceduren, `Form_Closing()`, är en procedur som anropas när formuläret håller på att stängas. I koden ovan så visas en meddelanderuta (med frågetecken och Ja/Nej-knappar) som frågar användaren om denne (eller denna ☺) vill avsluta programmet. Om användaren svarar Nej så avbryts händelsen Close.

Denna sista procedur kan ni fylla i signaturen för metoden manuellt eller så kan ni välja **(Overrides)** i listrutan Class Name (längst upp till vänster i kodfönstret – se bild nedan) och sen **OnClosing** i listrutan Method Name (till höger). Fyll därefter i koden för metoden.



Figur 3 - Implementerande av ärvd metod som överskuggas.

2.6.4.2 Testkör programmet

Kör programmet och lägg till 1-2 taxibilar respektive lastbilar och klicka på tredje knappen för att skriva ut objekten i listrutan.

2.6.4.3 Ny version av SkrivUt()

Vill ni veta vilken typ respektive bil är samt dess egna egenskaper så kan ni dubbelklicka på knappen Skriv ut bilar och typ samt lägga till nedanstående kod i proceduren `btnSkrivUtTyp_Click()`.

Observera att vi måste ”konvertera” till rätt datatyp för att kunna anropa metoder i klasserna som metoder definierats. Objektet måste alltså tilldelas till en variabel av den subklass som

objektet är för att kunna anropa metoder i den subklassen. Vi använder (lämpligen) metoden `CType()` för konverteringen.[15]

```
Private Sub btnSkrivUtTyp_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnSkrivUtTyp.Click
  'Dekl. Variabler för objekt - en för varje typ som anv. (IBil, Taxi, Lastbil)
  Dim objBil As IBil, objTaxi As Taxi, objLastbil As Lastbil
  Dim strTyp As String, strEgen As String

  'För varje objekt i vektorn...
  For Each objBil In colBilar  'Här hämtas referens till aktuellt objekt i vektor
    If TypeOf (objBil) Is Taxi Then  'Om typen är Taxi ...
      objTaxi = CType(objBil, Taxi)  '... konvertera o tilldela till Taxi-variabel
      strTyp = " Taxi: "
      strEgen = "Antal passagerare: " & objTaxi.AntalPassagerare.ToString()
    ElseIf TypeOf (objBil) Is Lastbil Then   '... eller om är Lastbil ...
      objLastbil = CType(objBil, Lastbil)    '... konv. o tilldela t. Lastbil-var.
      strTyp = " Lastbil: "
      strEgen = "Maxlast: " & objLastbil.Maxlast.ToString()
    End If

    lstBilar.Items.Add(objBil.Marke & ", " & objBil.Arsmodell.ToString() _
      & strTyp & strEgen)
  Next

  lstBilar.Items.Add(" ")
End Sub
```

2.6.5 Kommentar till arv och komponenter i Visual Basic.NET

I.o.m. .NET så behöver komponenter inte längre ärva från gränssnittet `IUnknown` (se sammanfattningen *Komponenter med COM*) – dom ärver istället från klassen `Object` (direkt eller indirekt). Och arv har blivit lättare – något mer naturligt, d.v.s. som i andra objektorienterade programmeringsspråk.

2.7 Mer om arv i VB.NET

Som nämnts tidigare så har VB.NET blivit 100% objektorienterat, bl.a. med stöd för gränssnitt, abstrakta klasser och arv av implementation (kod). Nedan behandlas bl.a. hur man gör abstrakta klasser i VB.NET och hur man ärver från dessa. I exempel nedan ersätts gränssnittet IBil från exempel ovan med den abstrakta klassen CBil.

2.7.1 Abstrakta klasser och arv

Ett skäl till att använda abstrakta klasser (jämfört med gränssnitt [*interface*]) är för att samla kod som är gemensamt för alla subklasser till den abstrakta klassen. T.ex. så har klasserna Lastbil och Taxi båda egenskaperna Märke och Årsmodell. För att slippa behöva dubblera koden (d.v.s. implementera egenskaperna i båda subklasser) så kan vi placera denna kod i klassen CBil. Men en Bil kan inte existera – klassen CBil är endast en abstraktion av klasser som Lastbil, Taxi och Buss, d.v.s. en Taxi är en Bil – därför görs klassen CBil abstrakt.

En abstrakt klass måste deklareras med det reserverade ordet `MustInherit` före och metoder som är abstrakta måste deklareras med det reserverade ordet `MustOverride` före. I exempel nedan är klassen abstrakt samt metoden `Kor()`.

Klasser som implementerare en abstraktklass måste sen ärva från den abstrakta klassen m.h.a. det reserverade ordet `Inherits`. Eventuella abstrakta metoder i den abstrakta klassen måste

[15] I VB.NET (liksom VB6) sker implicit konvertering om vi inte använder metoden `CType()`. Detta är dock inte vidare effektivt samt kan leda till fel och vi bör därför alltid använda explicit konvertering.

föregås av det reserverade ordet `Overrides`. I motsats till då vi implementerar ett gränssnitt så behöver vi inte ange vilken metod vi implementerar – vi döljer (överskuggar – *overrides*) ju superklassens metod med en ”konkret” metod (i motsats till abstrakt).

Nedan har gränssnittet IBil ersatts av en abstrakt klass CBil som sen klasserna Lastbil och Taxi ärver från. Nedan visas koden för den abstrakta klassen CBil.

```
'Deklarera klass som abstrakt med MustInherit
Public MustInherit Class CBil
  Private mstrMarke As String        'Deklarera instansvariabler för klass
  Private mintArsmodell As Integer

    'Implementera egenskaper i klass
  Public Property Marke() As String
    Get
      Return mstrMarke
    End Get
    Set(ByVal Value As String)
      mstrMarke = Value
    End Set
  End Property

  Public Property Arsmodell() As Integer
    Get
      Return mintArsmodell
    End Get
    Set(ByVal Value As Integer)
      mintArsmodell = Value
    End Set
  End Property

  'Deklarerar metod abstrakt med MustOverride
  Public MustOverride Function Kor() As String
End Class
```

Nedan visas koden för klassen Taxi och nedanför det koden för klassen Lastbil.

```
Public Class Taxi
  Inherits CBil

    'Deklarera endast instansvariabler för egna egenskaper - övriga ärvs
  Private mintAntalPassagerare As Integer

    'Implementera egna egenskaper i klass
  Public Property AntalPassagerare() As Integer
    Get
      Return mintAntalPassagerare
    End Get
    Set(ByVal Value As Integer)
      mintAntalPassagerare = Value
    End Set
  End Property

    'Implementera abstrakt metod genom att överskugga med Overrides
  Public Overrides Function Kor() As String
    Return "Trycker på gaspedalen (i Taxi)"
  End Function
End Class
```

```
Public Class Lastbil
  Inherits CBil

    'Deklarera endast instansvariabler för egna egenskaper - övriga ärvs
  Private mdblMaxLast As Double

    'Implementera egna egenskaper i klass
  Public Property AntalPassagerare() As Double
```

```
      Get
        Return mdblMaxLast
      End Get
      Set(ByVal Value As Double)
        mdblMaxLast = Value
      End Set
    End Property

      'Implementera abstrakt metod genom att överskugga med Overrides
    Public Overrides Function Kor() As String
      Return "Trycker på gaspedalen (i Lastbil)"
    End Function
  End Class
```

Om vi jämför koden ovan med den kod som krävdes då vi använde gränssnittet IBil så ser vi att implementationerna av klasserna Lastbil och Taxi är betydligt mycket kortare nu. Detta då det som är gemensamt för alla subklasser (d.v.s. Lastbil och Taxi) har placerats i klassen CBil.

Dessa nya klasser kan testas med samma kod som i förra klienten med mindre justeringar, bl.a. användandet av klassen CBil istället för gränssnittet IBil.

2.8 Klassbibliotek

I .NET kan vi numera även skapa s.k. klassbibliotek[16]. Ett klassbibliotek är egentligen bara en samling med klasser och gränssnitt som placerats i en mjukvarumodul (komponent om ni så vill ☺). Dessa klassbibliotek kan vara ”privata” (för en applikation) eller delade (för alla applikationer på dator – se kapitel *Assemblies*).

[16] Det som i VB6 motsvaras av en ActiveX DLL.

3 Felhantering

I VB.NET kan vi fortfarande (som i VB6) använda oss av `On Error Goto …`, vilket av en del kallas för **ostrukturerad felhantering**. Men vi har även möjlighet att använda oss av undantag (*exceptions*) och Try-Catch-block, vilket kallas **strukturerad felhantering**. En fördel med undantag är att vi (i t.ex. VB.NET) kan fånga fel som uppstått i programmoduler skrivna med andra programmeringsspråk (t.ex. C#) och vice versa. Detta är ett skäl till att vi bör sluta använda ostrukturerad felhantering till förmån för strukturerad felhantering.

3.1 Använda sig av felhanterare

Strukturerad felhantering i VB.NET bygger alltså på undantag samt Try-block med matchande Catch-block. I sin enklaste form så behöver vi endast Try- och Catch-blocken, men det finns varianter där t.ex. Catch-blocket kan utelämnas.

3.1.1 Enklaste formen av Try-Catch-block

```
Try
  'Kod som ska exekveras – om fel uppstår här, exekveras Catch-block
Catch
  'Kod som ska exekveras om fel i Try-blocket
End Try
```

3.1.2 Fånga specifika fel

För att kunna erhålla information om vilken typ av fel som uppstått (d.v.s. vilken typ av undantag som ”slängts”) så måste vi använda en ”undantagsparameter” i Catch-blocket.

```
Catch Exception_parameter As Exception_typ
  'Kod som ska exekveras om undantag av typen Exception_typ slängts i Try-blocket
End Try
```

Om vi vill kunna hantera fler än en typ av undantag så kan vi använda flera Catch-block med olika typer på våra ”undantagsparametrar”.

```
Catch Ex1 As Ex_typ1
  'Kod som ska exekveras om undantag av typen Ex_typ1 slängts i Try-blocket
Catch Ex2 As Ex_typ2
  'Kod som ska exekveras om undantag av typen Ex_typ2 slängts i Try-blocket
Catch Ex3 As Ex_typ3
  'Kod som ska exekveras om undantag av typen Ex_typ3 slängts i Try-blocket
End Try
```

3.1.3 Finally-block

Utöver Try- och Catch-blocken kan vi använda ett Finally-block, kod som vi vill ska exekvera oavsett om fel uppstår i Try-blocket eller inte.

```
Try
  'Kod som ska exekveras – om fel uppstår här, exekvera Catch-block
Catch
  'Kod som ska exekveras om fel i Try-blocket
Finally
  'Kod som ska exekveras oavsett om fel i Try-blocket
End Try
```

3.2 Generera ett fel

Att generera ett fel innebär att ”slänga ett undantag” (*throw an exception*). Nyttan med att kunna generera ett fel är för att kunna meddela klienten (av t.ex. vår klass) om att ett fel har uppstått eller för att kunna generera mer begripliga felmeddelande (än de som t.ex. .NET *data providers* ger).

För att generera ett fel skapar vi en ny instans av klassen `Exception` eller någon av dess subklasser samt använder det reserverade ordet `Throw`.

```
Throw New Exception(”Text som beskriver fel”)
```

Om vi vill skicka med ett fel som ledde till att vi genererade ett fel i vår kod så kan vi skicka felet (det första ☺) som andra parameter till konstruktorn för klassen `Exception`.

```
Try
  'Kod som genererar första felet
Catch OleE As OleDbException
  Throw New Exception(”Text som beskriver fel”, OleE)
End Try
```

3.2.1 Om namn på undantagsvariabel i Catch-block

Använd **inte** namnet `e` som namn på undantagsvariabel i Catch-block! I metoder som hanterar händelser (t.ex. när användare klickar på knappar) så heter variabel med händelse som skickas till händelsehanteraren (metoden) `e`. Om vår undantagsvariabel också heter `e` så döljer vi händelsevariabeln.

Nedanstående kod kommer alltså inte kompileras utan fel. Metoden är en händelsehanterare för knappen `btnUtanFelhantering`. Observera att andra parametern till metoden heter `e` och är av typen `System.EventArgs`.

```
Private Sub btnUtanFelhantering_Click(ByVal sender As System.Object, _
                  ByVal e As System.EventArgs) Handles btnUtanFelhantering.Click
  Try
    Utan_felhantering()
  Catch e As Exception  'Denna deklaration av undantagsvariabel kompilerar inte!
    MessageBox.Show(e.Message, "Fel", MsgBoxStyle.Critical)
  End Try
End Sub
```

4 Databasprogrammering med ADO.NET

I.o.m. ADO.NET så har Microsoft ändrat lite i sina databaskomponenter (d.v.s. från ”gamla hederliga” ADO version 2.x), men grundtanken är den samma: En förbindelse till en databas motsvaras av ett Connection-objekt, ett kommando (SQL-sats eller öppnande av tabell) utförs av ett Command-objekt och resultatet från kommandot placeras i ett objekt som representerar resultatet (Recordset-objektets ersättare).

Vi har bl.a. fått några nya objekt att arbeta med – DataReader och DataSet – objekt som ska ersätta Recordset-objektet från ADO. Skillnaden mellan dessa objekt är att den första används för att endast läsa data och den senare för att även uppdatera data. Ytterligare en skillnad är att ett DataSet-objekt kan innehålla resultat från flera tabeller och relationerna mellan tabellerna (vilket också gör det till ett komplext objekt ☺).

I ADO.NET använder vi något som kallas **.NET Data Providers** (”drivrutiner”) för att ansluta till datakällor. Med VS.NET levereras tre olika data providers: **SQL Server .NET Data Provider** för Microsoft SQL Server, **Microsoft .NET Data Provider for Oracle**[17] för Oracle och **OLE DB .NET Data Provider** för övriga datakällor.[18] I denna beskrivning kommer jag endast att behandla OLE DB-varianten, främst då den kan användas för alla databaser med drivrutiner för ADO, t.ex. Oracle, SQL Server och Access (se MSDN för beskrivning av SQL Server-varianten). Ytterligare en orsak är likheten mellan de flesta *data providers*, bl.a. då de ärver från samma superklasser. Den största skillnaden (bortsett från vilka drivrutiner de använder ☺) är att klasserna finns i olika namnutrymmen (t.ex. `System.Data.OleDb` och `System.Data.SqlClient`) och att klasserna har olika prefix (`OleDb`, `Ora` resp. `Sql`).

Det mesta av dataåtkomsten i ADO.NET bygger på att kontakt etableras med datakälla, fråga utförs och resultatet kopplas loss från datakällan (det som kallades *disconnected recordsets* i ADO). För att uppdatera eller lägga till data så ansluts till datakällan igen för att genast kopplas loss. I .NET använder *data providers* som standard poolning av förbindelser.[19] På detta sätt hoppas Microsoft att dataåtkomst ska effektiviseras då endast ett fåtal delade förbindelser med datakällor behövs.

4.1 ADO.NET-modellen

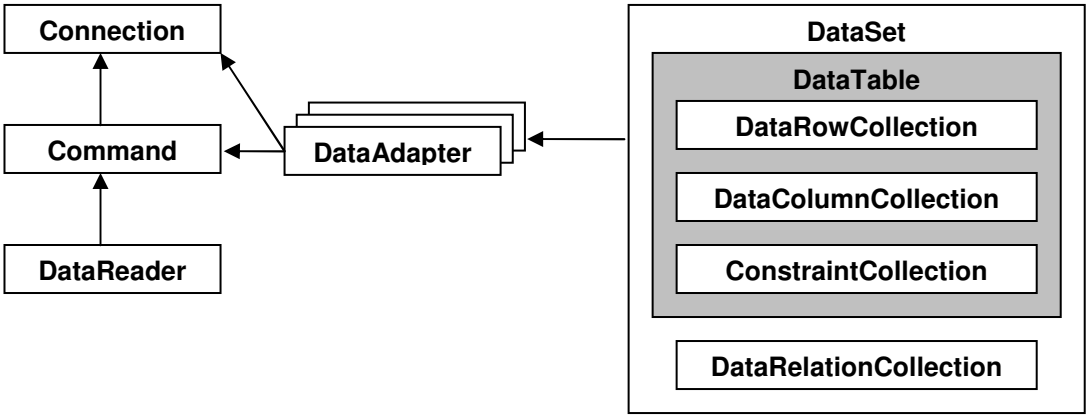
I ADO.NET heter klassen för våra Connection-objekt `OleDbConnection` (eller `SqlConnection` om SQL Server) och för våra Command-objektet `OleDbCommand` (resp. `SqlCommand`). Dessa två objekt används för att etablera kontakt med datakälla och utföra ett kommando (en SQL-sats, öppnandet av en tabell eller anropandet av en lagrad procedur). Resultatet av kommandot (om något) placeras i ett våra nya objekt i ADO.NET: vår DataReader för enbart läsning eller vårt DataSet för både läsning och uppdatering.

För att skapa ett DataSet-objekt från data i en datakälla så krävs ytterligare ett objekt: en DataAdapter. En DataAdapter är en brygga mellan vårt DataSet som innehåller data och datakällan. Det behövs en DataAdapter för varje tabell som våra DataSet innehåller, d.v.s. ett DataSet kan innehålla flera tabeller. Ett DataSet kan även innehålla relationerna (våra restriktioner, d.v.s. främmande nycklar) mellan tabellerna.

[17] Detta är Microsofts version av *data provider* för Oracle. Vi bör ladda ner Oracles egna version och använda den istället.
[18] Andra data providers kan laddas ner från Microsofts hemsidor eller tillverkaren av databasen.
[19] Förbindelsepoolning kräver att våra ConnectionString är lika för alla dataåtkomster (se *Connection-objekt*).

Klassen för en DataAdapter beror på vilken data provider som används. I OLE DB så heter klassen `OleDbDataAdapter` (och för SQL Server `SqlDataAdapter`). Klassen `DataSet` är den samma oavsett data provider och klassen finns i namnutrymmet `System.Data`.

I vår DataSet finns bl.a. två stycken vektorer som egenskaper: Tables och Relations. Dessa vektorer innehåller objekt av typen DataTable respektive DataRelation. Som det låter på namnen så är det klasser som representerar tabeller och relationer mellan tabellerna.



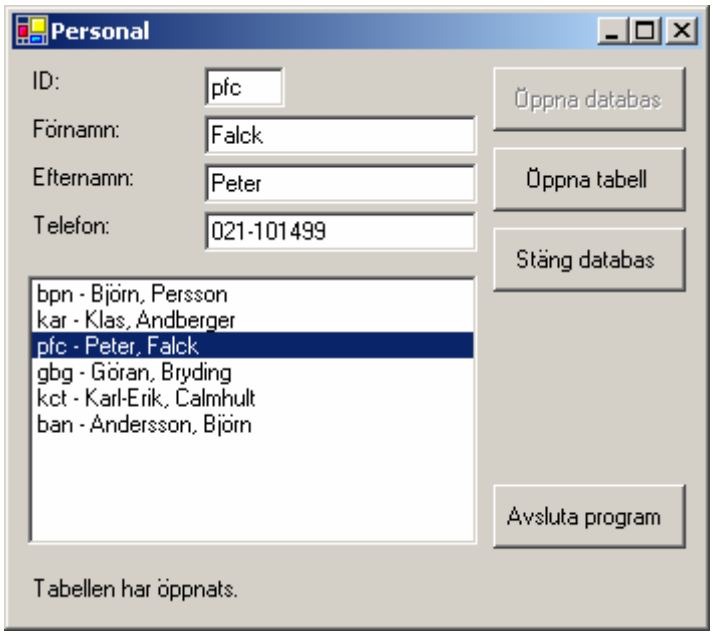
4.2 Ett första exempel

4.2.1 Tabell i Access och formulär i Visual Basic.NET

	Fältnamn	Datatyp	Beskrivning
▶		Text	Tre bokstävers förkortning
	FNamn	Text	Förnamn på person (20 tecken)
	ENamn	Text	Efternamn på person (30 tecken)
	Telefon	Text	Telefon vid MdH (15 tecken)

Figur 4 - Access-tabell som används i databasexempel.

I detta exempel kommer en tabell med namnet `tblPersonal` att användas. Tabellen har skapats i en Access-databas med namnet `BOKNING.MDB` (som ligger i roten på enhet C: – ändra i koden nedan om den ligger på annan plats) och har ovanstående design.



Formuläret (se bild till höger) innehåller

- fyra etiketter
- fyra textrutor (`txtSignatur`, `txtFornamn`, `txtEfternamn` och `txtTelefon`)
- en listruta (`ListBox1`)
- och fyra knappar

4.2.2 Beskrivning av exempel

I detta första exempel så öppnas Access-databasen när användaren klickar på knappen Öppna databas och tabellen `tblPersonal` när användaren klicka på knappen Öppna tabell. För varje

post i tabellen skapas en instans av klassen Personal (se nedan för definition av klassen) som sen läggs till i en listruta. (En nyhet med .NET är att vi numera kan placera objekt i t.ex. listrutor, d.v.s. inte bara strängar. Genom att lägga till objekt i listrutan så kan vi hämta värden från objekten när användaren klickar på alternativet i listrutan.)

Först börjar vi med att deklarera en global variabel[20] för vårt Connection-objekt. Detta görs då vi kommer att öppna förbindelsen till databasen i händelsehanteraren för en knapp samt öppna tabellen i en annan händelsehanterare. Vi passar även på att lägga till två konstanter för ConnectionString och SQL-sats (som vi använder prefixet ”c” för att tala om att det är konstanter). Klicka på knappen View Code i Solution Explorer (knappen längst till vänster i bild till höger) och lägg till följande rader längst upp i formulärets klass, under raden som börjar med `Inherits ….`


```
    'Global variabel för Connection-objekt
Private madoConn As OleDb.OleDbConnection

    'ConnectionString och SQL-sats att utföra (ändra ev. sökväg till databasfil)
Private Const cstrConn As String = "Provider=Microsoft.Jet.OLEDB.4.0;" _
        & "Data Source=C:\bokning.mdb;Persist Security Info=False"
Private Const cstrSQL As String = "SELECT * FROM tblPersonal"
```

Återgå till formuläret (genom att t.ex. klicka på knappen View Designer till höger om knappen View Code) och dubbelklicka på knappen Öppna databas för att skapa metodsignaturen för knappens händelsehanterare. Lägg sen till följande kod i den nyligen skapade händelsehanteraren. Här skapas ett Connection-objekt och ConnectionString bifogas till konstruktorn. Sen anropas metoden `Open()` för att öppna förbindelse till databas.

```
    'Skapa Connection-objekt och öppna förbindelsen
madoConn = New OleDb.OleDbConnection(cstrConn)
madoConn.Open()
```

Återgå (igen ☺) till formuläret och dubbelklicka på knappen Öppna tabell och lägg till följande kod i händelsehanterare som skapas. Först skapas ett Command-objekt och egenskaper för den sätts. Sen anropas metoden `ExecuteReader()` i Command-objekt för att hämta poster för enbart läsning i form av ett DataReader-objekt. DataReader-objektet används sen för att loopa över posterna. I loopen skapas en instans av personalobjekt som fylls med data från aktuell post. Eftersom telefonnummer kan vara NULL i databas så måste vi först testa om det inte är NULL innan vi hämtar värdet så att fel inte uppstår. Personalobjektet läggs sen till i listrutan. Sist av allt så stängs DataReader-objektet så att databasförbindelsen kan användas av andra.

```
Dim adoCmd As OleDb.OleDbCommand, adoReader As OleDb.OleDbDataReader
Dim objPersonal As Personal, objTemp As Object

    'Skapa Command-objekt och sätt egenskaper för objektet
adoCmd = New OleDb.OleDbCommand()
adoCmd.CommandText = cstrSQL              'SQL-sats att utföra
adoCmd.CommandType = CommandType.Text     'Typ av kommand (SQL-sats, d.v.s. text)
adoCmd.Connection = madoConn              'DB-förbindelse att använda

    'Kör fråga och placera resultatet i DataReader-objekt
adoReader = adoCmd.ExecuteReader
```

[20] Den ”globala” variabeln är faktist en instansvariabel i formulärets klass.

```
    'Så länge det finns fler poster - lägg till person i listruta
  While adoReader.Read
      'Skapa nytt Personalobjekt och fyll med data från aktuell post
    objPersonal = New Personal()
    objPersonal.ID = adoReader.GetString(0)
    objPersonal.ENamn = adoReader.GetString(1)
    objPersonal.FNamn = adoReader.GetString(2)

    objTemp = adoReader.Item("Telefon")   'Kan vara NULL i DB -> hämta som Object

    If Not IsDBNull(objTemp) Then         'Testa att telefon inte är NULL först..
      objPersonal.Telefon = objTemp       '... om inte - sätt värde i Personalobjekt
    End If

    ListBox1.Items.Add(objPersonal)       'Lägg till Personal-objekt i listruta
  End While

  adoReader.Close()                       'Stäng DataReader
```

Och sist av allt så återgår vi till formuläret för att dubbelklicka på knappen Stäng databas och lägger till följande rad i knappens händelsehanterare.

```
  madoConn.Close()    'Stäng förbindelse till databas
  madoConn = Nothing  'Förstör objekt
```

Om ni vill så kan ni även lägga till nedanstående rad i händelsehanteraren för knappen Avsluta, men den är inte viktig för databasåtkomstens funktioner. ☺

```
  Me.Close()  'Stäng formulär och avsluta därmed program
```

(I exempel ovan har koden för listrutans händelsehanterare för markering/klickning inte tagits med. Men principen för denna händelsehanterare är att vi hämtar personen, d.v.s. personalobjektet, som markerats från listan och fyller i textrutorna med data från objektet.)

4.2.3 Klassen Personal

Klassen Personal är inte så avancerad – den innehåller endast fyra egenskaper som motsvarar kolumnerna i tabellen `tblPersonal`. Klasser som Personal kallas bl.a. för värdeobjekt (*value objects*), eller hjälpklasser, då dessa kan användas för att skicka data mellan presentation och affärslogik i distribuerade applikationer. Nedan har även metoden `ToString()` implementeras för att visa data om person när instans av klassen placeras i listrutan.

```
  Public Class Personal
    Private mstrID As String        'Deklarera variabler för klassens egenskaper
    Private mstrENamn As String
    Private mstrFNamn As String
    Private mstrTelefon As String

    Property ID() As String         'Definiera accessmetoder för klassens egenskaper
      Get
        Return mstrID
      End Get
      Set(ByVal Value As String)
        mstrID = Value
      End Set
    End Property

    Property ENamn() As String
      Get
```

```
        Return mstrENamn
      End Get
      Set(ByVal Value As String)
        mstrENamn = Value
      End Set
    End Property

    Property FNamn() As String
      Get
        Return mstrFNamn
      End Get
      Set(ByVal Value As String)
        mstrFNamn = Value
      End Set
    End Property

    Property Telefon() As String
      Get
        Return mstrTelefon
      End Get
      Set(ByVal Value As String)
        mstrTelefon = Value
      End Set
    End Property

      'Metod som ärvs från Object och som omdefinieras i denna klass
    Public Overrides Function ToString() As String
      Return mstrID & " - " & mstrENamn & ", " & mstrFNamn
    End Function

  End Class
```

5 Objekten i ADO.NET

Detta kapitel innehåller förklaringar till de (mest?) intressanta egenskaperna i objekten i ADO.NET. Detta kapitel är tänkt att vara en referens, d.v.s. bör inte läsas från första till sista sidan utan skummas för en överblick. Exempel kommer visas med VB.NET-kod, men fungerar på liknande sätt i andra .NET-språk.

Nästa kapitel innehåller databasfunktioner (hämta, infoga, uppdatera och radera) i kod.

5.1 Objektet OleDbConnection

Ett OleDbConnection-objekt (eller bara Connection-objekt) motsvarar en förbindelse (session) mot datakälla. Innan vi kan utföra några kommando mot datakälla måste vi alltså etablera en förbindelse mot datakälla genom att skapa ett Connection-objekt.

Användbara egenskaper och metoder i objekt av typen Connection är:

- Konstruktor utan parametrar
- Konstruktor med ConnectionString som parameter
- .ConnectionString
- .Open()
- .Close()
- .CreateCommand()

En skillnad mellan ADO och ADO.NET är att i ADO.NET så finns ingen Execute()-metod i Connection-objektet. Men vi kan använda `CreateCommand()` för att erhålla ett Command-objekt som kopplats till vårt Connection-objekt. I Command-objektet kan vi sen anropa `Execute()`.

5.1.1 Konstruktor utan parametrar

Om vi använder konstruktorn utan parametrar så måste vi ange värde för egenskapen ConnectionString (se nedan) innan vi kan öppna datakällan.

```
Dim adoConn As OleDb.OleDbConnection

  'Skapa ny instans - använd konstruktor utan parameter
adoConn = New OleDb.OleDbConnection()
```

5.1.2 Konstruktor med ConnectionString som parameter

Genom att skicka vår ConnectionString som parameter till konstruktorns så kan vi direkt anropar metoden `Open()` för att öppna förbindelsen med datakällan.

```
Dim adoConn As OleDb.OleDbConnection
Dim strConn = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=..."

  'Skapa ny instans - använd konstruktor med parameter
adoConn = New OleDb.OleDbConnection(strConn)
```

5.1.3 Egenskapen ConnectionString

Denna egenskap är i stort sett samma som egenskapen ConnectString i ADO, d.v.s. vi kan t.ex. fortfarande använda en UDL-fil för att skapa en ConnectionString.

```
adoConn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" _
```

```
                    & "Data Source=C:\bokning.mdb;Persist Security Info=False"
```

5.1.4 Metoden Open

Fungerar som i ADO – öppnar förbindelse till datakälla. Skillnaden mot ADO är att denna metod inte tar några parametrar. Innan metoden kan anropas måste ConnectionString angivits – antingen via egenskapen ConnectionString eller när Connection-objekt skapades.

```
adoConn = New OleDb.OleDbConnection(strConn)
adoConn.Open()  'Öppna förbindelse till datakälla
```

5.1.5 Metoden Close

Fungerar som i ADO – stänger förbindelse till datakälla.

```
adoConn.Close()  'Stäng förbindelse till datakälla
```

5.1.6 Metoden CreateCommand

Metoden CreateCommand skapar en instans av klassen OleDbCommand med dess egenskap .Connection satt till aktuellt Connection-objekt (se nästa avsnitt).

```
Dim adoCmd As OleDb.OleDbCommand
  '...
adoCmd = adoConn.CreateCommand()  'Skapa Command-objekt
```

5.2 Objektet OleDbCommand

Command-objektet används för att öppna en tabell, utföra en SQL-sats eller anropa en lagrad procedur (*stored procedure*), d.v.s. utföra ett kommando. Kommandot utförs genom att anropa någon av de tre Execute-metoderna som returnerar, om något, ett objekt (DataReader eller enstaka värde) som innehåller resultatet från kommando (se även avsnitten *Objektet OleDbDataAdapter* och *Objektet DataSet*). Command-objekt kräver en förbindelse till en datakälla, d.v.s. ett Connection-objekt.

Användbara egenskaper och metoder i objekt av typen OleDbCommand är:

- Konstruktor utan parametrar
- Konstruktor med CommandText och/eller Connection-objekt som parameter
- .CommandText
- .CommandType
- .Connection
- .Parameters
- .CreateParameter()
- .ExecuteNonQuery()
- .ExecuteReader()
- .ExecuteScalar()

Command-objektet har fyra konstruktorer, varav endast tre behandlas nedan (den med transaktion utelämnas då den inte behövs för komponenter som installeras i MTS/COM+). Vi kan även använda metoden `CreateCommand()` i Connection-objektet för att skapa en instans av klassen (se föregående avsnitt). Vi bör alltid ange värde på egenskapen `CommandType` oavsett hur Command-objekt skapas (se nedan).

5.2.1 Konstruktor utan parametrar

Om vi skapar en instans av klassen utan att skicka någon parameter till konstruktorn så får vi en ny instans av klassen. Vi måste då använda egenskapen `Connection` (se nedan) för att ange vilken förbindelse (Connection-objekt) som ska användas samt egenskapen `CommandText` för att ange vilket kommando som ska utföras (SQL-sats, öppna tabell eller anropa en lagrad procedur).

```
Dim adoCmd As OleDb.OleDbCommand

  'Skapa instans - använd konstruktor utan parametrar
adoCmd = New OleDb.OleDbCommand()
```

Alternativt kan vi använda metoden `CreateCommand()` i Connection-objektet och endast ange egenskapen `CommandText`. Eller så kan vi skicka kommando och Connection-objektet som parameter till konstruktorn (se nedan).

5.2.2 Konstruktor med CommandText och/eller Connection-objekt som parameter

Utöver konstruktorn utan parametrar finns tre till (varav endast två behandlas här) med följande parametrar:

- CommandText
- CommandText och Connection-objekt

5.2.2.1 Konstruktor med en parameter

Använder vi den första varianten så skickas SQL-satsen (som en sträng) som parameter. Connection-objekt måste anges med egenskapen `Connection` (se nedan).

```
Dim adoCmd As OleDb.OleDbCommand

  'Skapa instans - använd konstruktor med en parameter
adoCmd = New OleDb.OleDbCommand(strSQL)
```

5.2.2.2 Konstruktor med två parametrar

Om vi istället använder den andra varianten (med 2 parametrar) så skickar vi SQL-satsen och Connection-objektet som ska användas som parametrar till konstruktorn. Vi kan då utföra kommandot direkt efter att ha skapat objektet.

```
Dim adoCmd As OleDb.OleDbCommand

  'Skapa instans - använd konstruktor med två parametrar
adoCmd = New OleDb.OleDbCommand(strSQL, adoConn)
```

5.2.3 Egenskapen CommandText

Egenskapen `CommandText` är något av följande:

- SQL-sats som ska utföras
- tabell som ska öppnas

- eller lagrade frågan (*stored procedure*) som ska anropas.

```
adoCmd.CommandText = "SELECT * FROM tblPersonal"
```

Istället för att ange denna egenskap separat så kan vi skicka SQL-satsen som parameter till konstruktorn (se ovan).

5.2.4 Egenskapen CommandType

Anger hur egenskapen `CommandText` (se ovan) ska tolkas. Lämpligen används någon av konstanterna nedan (vilka har definierats i uppräkningstypen `CommandType` i `System.Data`).

- `CommandType.Text` – (standard) `CommandText` innehåller en SQL-sats.
- `CommandType.TableDirect` – `CommandText` innehåller namnet på en tabell.
- `CommandType.StoredProcedure` – `CommandText` innehåller en lagrad procedur.

```
adoCmd.CommandType = CommandType.Text
```

5.2.5 Egenskapen Connection

Connection-objekt som ska användas som förbindelse till datakälla – motsvarar egenskapen ActiveConnection i ADO. (Vi behöver inte längre använda det reserverade ordet `Set` för att ange objekt för denna egenskap som i VB6.)

```
adoCmd.Connection = adoConn
```

Alternativt kan Connection-objekt skickas som parameter till konstruktorn (se ovan) eller Command-objekt skapas med metoden `CreateCommand()` i Connection-objekt (se tidigare avsnitt om Connection-objekt).

5.2.6 Vektorn Parameters

Vektorn `Parameters` används för att lagra alla eventuella parametrar (t.ex. frågetecken) som finns i en SQL-sats eller för att skicka in- och utparametrar till en lagrad procedur (*stored procedure*). (Se nästa avsnitt för beskrivning av objektet Parameter.) Parameter-objekt kan skapas med metoden `CreateParameter()` (se nedan) eller som vilket objekt som helst (d.v.s. med `New` ☺).

Parametrar bör skapas och läggas till i den ordning som de förekommer i SQL-sats eller den lagrade procedurens parameterlista (om de inte namnges – se nästa avsnitt).

```
Dim adoCmd as OleDb.OleDbCommand
Dim adoParam as OleDb.OleDbParameter

  'SQL-sats med en parameter (frågetecknet)
adoCmd.CommandText = "SELECT * FROM tblPersonal WHERE Id = ?"

adoParam = adoCmd.CreateParameter()      'Skapa ett Parameter-objekt

  'Ange egenskaper för Parameter-objekt...

adoCmd.Parameters.Add(adoParam)        'Lägg till Parameter-objekt i vektorn
```

5.2.7 Metoden CreateParameter

I motsats till metoden `CreateParameter()` i ADO:s Command-objekt så tar inte denna metod några parametrar. Metoden returnerar en instans av klassen Parameter som vi sen måste ange egenskaperna för (se nästa avsnitt).

```
Dim adoCmd as OleDb.OleDbCommand
Dim adoParam as OleDb.OleDbParameter

adoParam = adoCmd.CreateParameter()  'Skapa ett Parameter-objekt
```

5.2.8 Metoden ExecuteNonQuery

Utför en SQL-sats som inte returnerar några poster, t.ex. SQL-sats med UPDATE eller INSERT. Metoden returnerar ett heltal (`Integer`) med antal poster som påverkades av SQL-satsen (som med andra funktioner i VB.NET så behöver vi inte fånga upp värdet i en variabel om vi inte vill).

Innan denna metod kan anropas så måste egenskapen `CommandText` och `Connection` ha angivits, d.v.s. kommando som ska utföras och i vilken datakälla.

```
Dim intAntal As Integer

intAntal = adoCmd.ExecuteNonQuery()
```

5.2.9 Metoden ExecuteReader

Utför kommando (i egenskapen CommandText) som returnerar en instans av klassen `DataReader`. Om kommando är en SQL-sats så måste det vara av typen SELECT.

Egenskapen Connection och CommandText (samt eventuella parametrar) måste ha angivits först innan denna metod kan anropas.

```
Dim adoReader as OleDb.OleDbDataReader

adoCmd.CommandText = "SELECT * FROM tblPersonal" 'Ange SQL-sats att utföra
adoReader = adoCmd.ExecuteReader()               'Utför SQL-sats
```

Metoden kan även ta en parameter av typen (uppräkningstypen) `CommandBehavior` för att avgöra hur Connection-objektet ska agera. Vi kan t.ex. skicka `CommandBehavior.CloseConnection` som parameter för att även Connection-objektet ska stängas när vi stänger vår `DataReader` (vilket rekommenderas[21]).

```
Dim adoReader as OleDb.OleDbDataReader

adoCmd.CommandText = "SELECT * FROM tblPersonal"         'Ange SQL-sats att utföra
adoReader = adoCmd.ExecuteReader(CommandBehavior.CloseConnection)  'Utför SQL-sats
  'Manipulera data i DataReader
adoReader.Close  'Stäng även Connection-objekt
```

[21] Om vi inte använder `CommandBehavior.CloseConnection` så finns det ingen garanti när databasförbindelse stängs om vi t.ex. returnerar ett DataReader-objekt från en komponent. Förbindelse stängs då först när *garbage collection* utförs.

Observera att förbindelse till databasen som standard hålls öppen så länge som DataReader-objektet är öppet! Använd därför endast en `DataReader` för att t.ex. hämta alla poster och skriva ut med en gång. Stäng DataReader-objektet så fort som möjligt.

Observera även att den enda tillåtna operationen på Connection-objektet är att stänga den (genom att anropa metoden Close i Connection-objektet) under tiden som DataReader-objektet är ”aktivt”. D.v.s. vi kan inte använda Connection-objektet för t.ex. ytterligare dataåtkomst så länge DataReader-objektet är öppet.

5.2.10 Metoden ExecuteScalar

Utför kommando (SQL-sats eller lagrad fråga) samt returnerar resultatet från första kolumnen i första posten som datatypen Object. Resterande kolumner och poster ignoreras.

```
Dim obj As Object, intMax As Integer

adoCmd.CommandText = "SELECT MAX(Pris) FROM tblProdukter"    'Ange sats att utföra
obj = adoCmd.ExecuteScalar()                                 'Utför SQL-sats
intMax = CInt(obj)
```

5.3 Objektet OleDbParameter

Parameter-objektet används för att representera parametrar (t.ex. frågetecken – ?) i en SQL-sats eller lagrad procedur samt för returvärden från lagrad fråga. D.v.s. vi använder ett Parameter-objekt för både in-/utparametrar samt returvärden. Objektet är främst intressant tillsammans med ett Command-objekt (se föregående avsnitt).

Klassen innehåller främst konstruktorer och egenskaper av intresse:

- Konstruktor utan parametrar
- Konstruktor med parametrar (5 varianter, varav 1 behandlas här)
- .Direction
- .OleDbType
- .ParameterName
- .Size
- .SourceColumn
- .Value

5.3.1 Konstruktor utan parametrar

Skapar en instans av objektet…☺ Använder vi denna konstruktor när objektet skapas så måste egenskapen `Value` anges om det är en parameter i en SQL-sats eller inparameter till en lagrad procedur.

```
Dim adoParam As OleDb.OleDbParameter

  'Skapa instans – använd konstruktor utan parametrar
adoParam = New OleDb.OleDbParameter()
```

Alternativ till att skapa Parameter-objekt med denna standardkonstruktor är att använda metoden `CreateParameter()` i Command-objektet (se föregående avsnitt).

5.3.2 Konstruktor med parametrar

Klassen Parameter har 5 konstruktorer med parametrar (endast 1 behandlas här). Alla konstruktorer har namnet för Parameter-objektet (egenskapen `ParameterName` – se nedan) som parameter till sig, men vi kan om vi vill skicka en tom sträng.

Konstruktor som behandlas nedan har följande parametrar:

- ParameterName och Value

Om vi använder varianten av konstruktorn angiven här så bifogar vi värdet på parametrarna – övriga egenskaper kan oftast *data provider* ”räkna ut” själv.

```
Dim adoParam As OleDb.OleDbParameter
Dim intAntal As Integer

intAntal = 5

  'Skapa instans – använd konstruktor utan parametrar
adoParam = New OleDb.OleDbParameter("Antal", intAntal)
```

5.3.3 Egenskapen Direction

Anger vilken riktning (in, ut, in/ut eller returvärde) som parameter ska skickas. Denna egenskap bör användas för att ange parameters riktning då *data provider* kan optimera användning av parameter.[22] Lämpligen används något av värdena i uppräkningstypen `ParameterDirection` för att ange riktning.

- `ParameterDirection.Input` – värde skickas via parameter (kommer ej ändras).
- `ParameterDirection.InputOutput` – värde skickas och returneras via parameter.
- `ParameterDirection.Output` – värde returneras via parameter (kan ändras).
- `ParameterDirection.ReturnValue` – ett returvärde från t.ex. en lagrad funktion.

I motsats till Command-objekt i ADO så kommer inte Command-objekt i ADO.NET att returnera några poster om minst ett Parameter-objekt har egenskapen `Direction` satt till `ParameterDirection.Output`.[23]

```
adoParam.Direction = ParameterDirection.Input
```

5.3.4 Egenskapen OleDbType

För att *data provider* ska kunna optimera dataåtkomst bör man ange datatyp för parameter. Nedan visas några exempel på värden i uppräkningstypen `OleDbType` (som definierats i namnutrymme `System.Data.OleDb`). (Sök på ”OleDbType enumeration” i hjälpen för komplett lista på värden för denna egenskap.)

- `OleDbType.Boolean` – ett booleskt värde.
- `OleDbType.Char` – sträng med fast längd (icke-Unicode).
- `OleDbType.Date` – datum lagrad som Double.
- `OleDbType.Decimal` – decimaltal.
- `OleDbType.Integer` – heltal.
- `OleDbType.VarChar` – sträng med variabel längd (icke-Unicode).
- `OleDbType.VarWChar` – (standard) sträng med variabel längd (Unicode).

[22] *Data provider* behöver t.ex. inte skicka något värde till datakälla för en utparameter, d.v.s. sparar tid och plats.
[23] Jag är osäker om detta stämmer…

Denna egenskap är ”sammanlänkad” med egenskapen `DbType`, d.v.s. ändras en av dessa egenskaper så ändras även den andra. Ett skäl till att använda egenskapen `DbType` istället för `OleDbType` är för att göra kod som enklare kan översättas om datakälla ersätts.

```
adoParam.OleDbType = OleDbType.Integer
```

5.3.5 Egenskapen ParameterName

Denna egenskap är främst användbara om *data provider* stödjer namn på parametrar (t.ex. de för Jet/Access och SQL Server). Denna egenskap kan också vara användbar om man inte vill använda index för åtkomst av parametrar i Command-objektets vektor `Parameters`.

```
adoParam.ParameterName = "Antal"
```

5.3.6 Egenskapen Size

Denna egenskap är främst användbar om parameterns värde är en sträng av variabel längd (eller binärt). Om denna egenskap inte anges så kommer *data provider* att ta reda på värdets längd själv (till en viss kostnad).

```
strNamn = "Björn"
adoParam.Value = strNamn
adoParam.Size = strNamn.Length()

adoParam.Size = "Björn".Length()  'Följande fungerar faktiskt... ☺
```

5.3.7 Egenskapen SourceColumn

Denna egenskap är främst relevant om man använder ett DataSet-objekt. Egenskapen kan t.ex. användas för att hämta namnet på kolumn som parameter motsvarar i DataSet-objekt.

```
adoParam.SourceColumn = "Fnamn"
```

5.3.8 Egenskapen Value

Detta är den viktigaste egenskapen för ett Parameter-objekt ☺. Den används för att ange värdet på en inparameter och för att hämta värdet från en utparameter.

```
adoParam.Value = "Björn"
```

5.4 Objektet OleDbDataReader

OleDbDataReader-objekt motsvarar (nästan) ett Recordset-objekt i ADO där man använt konstanten `adForwardOnly`[24] för att utföra en SQL-sats (eller öppna en tabell). D.v.s. data i en DataReader kan endast läsas och **inte** ändras. För att kunna ändra data måste man använda Command- eller DataSet-objekt (se avsnitten nedan om objekten DataAdapter och DataSet).

- .FieldCount
- .GetOrdinal()

[24] Observera att konstanterna `adForwardOnly`, m.fl. inte finns i VB.NET! Vi använder som sagt en DataReader för att läsa från första till sista post och en DataSet för övrig dataåtkomst.

- .Item
- .Close()
- .GetXxxx()
- .GetName()
- .GetValues()
- .IsDBNull()
- .Read()

Klassen har inga konstruktorer då den skapas med metoden `ExecuteReader()` i Command-objekt.

Det är viktigt att anropa metoden `Close()` (d.v.s. ”stänga” objektet) så fort som möjligt eftersom så länge som objektet är ”öppet” så blockeras databasförbindelsen. Det är lika viktigt att öppna databaser och tabeller så sent som möjligt.

5.4.1 Egenskapen FieldCount

`FieldCount` innehåller antalet kolumner som resultatet eller tabellen innehåller. Denna egenskap kan endast läsas och är användbar för att loopa över kolumner i poster.

```
Dim adoReader As OleDb.OleDbDataReader
Dim intAntalKol As Integer

intAntalKol = adoReader.FieldCount()
```

5.4.2 Egenskapen Item

Egenskapen `Item` fungerar som en funktion och returnerar värdet i kolumn som skickas som parameter till egenskapen. Parametern kan vara antingen ett tal eller en sträng. I det första fallet så motsvaras talet med vilken kolumn (ordning) som värdet ska returneras från (första index är 0). Om vi istället skickar en sträng så ska det vara namnet på en kolumn i tabell eller frågeresultat.

```
strSignatur = adoReader.Item(0)           'Returnerar värdet i första kolumnen
strSignatur = adoReader.Item("Signatur")  'Returnerar värdet i kolumnen Signatur
```

Egenskapen returnerar värdet från kolumnen som typen Object, d.v.s. värdet måste konverteras (implicit eller explicit). Detta gör att detta sätt att hämta data från en datakälla inte är det mest effektiva. (Se *Metoderna GetXxxx* nedan.)

5.4.3 Metoden Close

Stänger DataReader. Detta frigör bl.a. Connection-objektet så att det kan användas till andra dataaccesser (om det inte stängs samtidigt som DataReader – se avsnitt *Metoden ExecuteReader* ovan).

```
adoReader.Close()
```

5.4.4 Metoderna GetXxxx

DataReader innehåller ett antal Get-metoder för att returnera data från datakällan **utan** konvertering (i motsats till egenskapen `Item` – se *Egenskapen Item* ovan).

Exempel på några av metoderna visas nedan. **Observera** att vi bör kontrollera om värdet i kolumnen är NULL **innan** vi anropar dessa Get-metoder – annars genereras ett fel (se

Metoden IsDBNull nedan). Nedan visas exempel på några metoder (sök på ”OleDbDataReader class” för komplett lista på alla metoder).

- `GetBoolean()` – returnerar ett booleskt värde.
- `GetDouble()` – returnerar ett decimaltal (`Double`).
- `GetInt32()` – returnerar ett heltal (motsvarande datatypen `Integer`).
- `GetInt64()` – returnerar ett heltal (motsvarande datatypen `Long`).
- `GetString()` – returnerar en sträng.

```
Dim strFnamn As String

strFnamn = adoReader.GetString("Fnamn")
```

5.4.5 Metoden GetName

Returnerar namnet på kolumnen vars ordningsnummer skickas som parameter.

```
Dim strKolNamn As String

strKolNamn = adoReader.GetName(0)           'Returnerar namnet på kolumn 0
```

5.4.6 Metoden GetOrdinal

Returnerar ordningsnummer för kolumn vars namn skickas som parameter till metoden.

```
Dim intKolNr As Integer

intKolNr = adoReader.GetOrdinal("Signatur")  'Returnerar nummer på kol. "Signatur"
```

5.4.7 Metoden GetValues

Metoden hämtar alla värden i en post som en vektor. När vi anropar metoden så måste vi bifoga en vektor (en vektor av typen `Object`) som parameter till metoden. Som returvärde från metoden returneras antalet positioner i vektorn som fyllts. Storleken (längden) på vektorn avgör hur många av värdena som kan hämtas, d.v.s. om vi deklarerar en vektor med för få positioner så kan inte alla värden (kolumner) i post returneras.

I exempel nedan deklareras en vektor av storleken 4 för att kunna ta emot fyra värden (d.v.s. från fyra kolumner) och antalet värden som vektor fylls med placeras i variabeln `intAntal`.

```
Dim arrObj(3) As Object                    'Vektor av storlek 4 (index 0 till 3)
Dim intAntal As Integer

intAntal = adoReader.GetValues(arrObj)     'Hämta värden och returnera antal värden
```

5.4.8 Metoden IsDBNull

Metoden tar nummer på kolumn som parameter och returnera sant om kolumn innehåller NULL i databasen. (Om vi inte vet nummer på kolumn så kan vi använda metoden `GetOrdinal()` – se ovan.)

Denna metod är användbar för att testa om ett värde är NULL i databasen innan vi använder någon av metoderna GetXxxx() (se ovan).

```
If Not adoReader.IsDBNull(0) Then        'Om kolumn 0 inte är NULL...
  '... hämta värde från kolumn
End If

If Not adoReader.IsDBNull(adoReader.GetOrdinal("Signatur")) Then  'Om inte NULL...
  '... hämta värde från kolumn
End If
```

5.4.9 Metoden Read

Flyttar postpekaren till nästa post och returnerar sant om det finns en nästa post, d.v.s. att postpekaren inte flyttas bortom sista posten.

Denna metod är användbar för att loopa över alla poster i aktuell DataReader.

```
While(adoReader.Read())           'Returnerar sant om det finns en post (till)
  'Skiv ut data från aktuell post
End While
```

5.5 Objektet OleDbDataAdapter [SKRIV OM]

Objekt av typen DataAdapter innehåller databasförbindelse och (minst) ett kommando, d.v.s. ett Connection-objekt och ett Command-objekt. En DataAdapter används för att koppla samman data med en förbindelse till datakälla (ett Connection-objekt) samt kommando som använts för att hämta data från datakälla (ett Command-objekt). Data som hämtats lagras i ett DataSet-objekt (se nästa avsnitt).

Nedan listas egenskaper och metoder av intresse i klassen OleDbDataAdapter.

- Konstruktor utan parametrar
- Konstruktor med parametrar
- .SelectCommand
- .DeleteCommand, .InsertCommand, och.UpdateCommand
- .Fill()
- .Update()

5.5.1 Konstruktor utan parametrar

Skapar en instans av DataAdapter.

```
Dim adoDA As OleDb.OleDbDataAdapter

adoDA = New OleDb.OleDbDataAdapter()
```

Om vi använder konstruktorn utan parametrar så måste vi ange värdet för egenskapen `SelectCommand` (och även de andra Command-egenskaperna om vi vill uppdatera data i datakällan). Värdet för egenskapen `SelectCommand` måste då vara ett Command-objekt som kopplats till ett Connection-objekt. (Se även *Objektet CommandBuilder* sist i detta avsnitt.)

5.5.2 Konstruktor med parametrar

Det finns tre varianter på konstruktor med parametrar och parametrarna listas nedan.

- Command-objekt
- SelectCommand (sträng) och Connection-objekt
- SelectCommand (sträng) och en ConnectionString (sträng)

Command-objekt skapade med konstruktorer nedan sätter bara (om alls) egenskapen `SelectCommand`. Vill vi även kunna uppdatera data i datakällan så måste vi ange värden för den övriga Command-egenskaperna separat (se DeleteCommand, InsertCommand och UpdateCommand nedan).

5.5.2.1 Konstruktor med Command-objekt

Använder vi konstruktorn med bara en parameter så anger vi värdet för egenskapen `SelectCommand` (se nedan). Command-objektet bör redan vara kopplat till en datakälla, d.v.s. dess egenskap Connection satt.

```
Dim adoCmd As OleDb.OleDbCommand
Dim adoDA As OleDb.OleDbDataAdapter

  'Skapa instans - skicka Command-objekt som redan kopplats till Connection-objekt
adoDA = New OleDb.OleDbDataAdapter(adoCmd)
```

5.5.2.2 Konstruktor med SelectCommand (sträng) och Connection-objekt

Om vi istället använder andra varianten av konstruktor (med parametrar) så ska vi skicka en sträng med kommando (motsvarande egenskapen `CommandText` i Command-objektet) och en dataförbindelse (d.v.s. ett Connection-objekt).

```
Dim adoConn As OleDb.OleDbConnection
Dim strSQL As String
Dim adoDA As OleDb.OleDbDataAdapter

  'Skapa instans - skicka SQL-sats (kommando som sträng) och Connection-objekt
adoDA = New OleDb.OleDbDataAdapter(strSQL, adoConn)
```

5.5.2.3 Konstruktor med SelectCommand (sträng) och ConnectionString (sträng)

Till sista varianten av konstruktorn skickar vi två strängar: ett kommando (motsvarande `CommandText` i Command-objekt) och en `ConnectionString` (i Connection-objekt).

```
Dim strSQL, strConn As String
Dim adoDA As OleDb.OleDbDataAdapter

  'Skapa instans - skicka SQL-sats (kommando) och ConnectionString (båda strängar)
adoDA = New OleDb.OleDbDataAdapter(strSQL, strConn)
```

5.5.3 Egenskapen SelectCommand [KONTROLLERA / EX.]

Denna egenskap kan anges som ett Command-objekt eller en sträng (?) med kommand (SQL-sats, tabellnamn eller lagrad procedur).

Detta är egentligen den enda egenskapen av Delete-, Insert-, Select- eller UpdateCommand som behöver anges. Vi kan använda en instans av typen CommandBuilder för att generera värden för de andra tre egenskaperna (se exempel senare).

5.5.4 Egenskaperna DeleteCommand, InsertCommand och UpdateCommand [KONTROLLERA / EX.]

Dessa egenskaper kan anges som ett Command-objekt eller en sträng (?) med kommando (SQL-sats, tabellnamn eller lagrad procedur). Värden för egenskaperna kan, i vissa fall,

genereras med ett CommandBuilder-objekt (se nedan). (Alternativet med CommandBuilder-objekt rekommenderas.)

5.5.5 Metoden Fill

Denna metod använder vi för att fylla ett DataSet-objekt (eller DataTable-objekt – se nästa två avsnitt) med data från datakällan. Metoden finns i 6 varianter, varav endast 2 behandlas här.

- DataSet
- DataSet och namn på tabell (sträng)

Vi kan även ersätta DataSet med ett DataTable-objekt (vilket inte behandlas här).

5.5.5.1 Metoden Fill med DataSet-objekt som parametrar

Innan vi kan använda data i ett DataSet-objekt så måste vi etablera kontakt med datakälla, skapa ett kommandoobjekt, skapa ett DataSet-objekt samt koppla samma dessa tre objekt. För detta använder vi en DataAdapter. Att koppla samman Connection- och Command-objekten använder vi DataAdapter-objektets egenskaper. Men för att kopplar samman DataSet-objektet använder vi metoden `Fill()`.

```
Dim adoConn As OleDb.OledDbConnection
Dim adoDA As OledDb.OledDbDataAdapter
Dim adoDS As Data.DataSet

  'Skapa instans av klasserna (tilldelning av egenskaper visas inte)
adoConn = New OledDB.OleDbConnection(strConn)
adoDA = New OleDb.OleDbDataAdapter(strSql, adoConn)
adoDS = New Data.DataSet()

  'Fyll DataSet med data från datakälla – utan att ange namn på tabell
adoDA.Fill(adoDS)
```

Om denna variant av metoden `Fill()` används så kan tabellerna endast refereras till (i DataSet-objekt) med nummer (se avsnitt nedan).

5.5.5.2 Metoden Fill med DataSet-objekt och sträng som parametrar

För att kunna referera till tabeller (i DataSet-objekt) med namn så kan vi skicka ytterligare en parameter till metoden `Fill()`, d.v.s. namn som vi vill kunna referera till tabell med (se avsnitt nedan).

```
   'Deklarera variabler och skapa instanser av objekt enligt ovan...

   'Fyll DataSet med data från datakälla – ange namn på "tabell"
adoDA.Fill(adoDS, "tblPersonal")
```

5.5.6 Metoden Update [UTVECKLA]

För att göra ändringar i DataSet beständiga i datakälla används denna metod. Vår DataAdapter måste först etablera en förbindelse med datakälla (igen!), utföra uppdateringar (m.h.a. ”Command-egenskaperna” ovan) gjorda i alla DataSet-objekt (relaterade/kopplade till DataAdapter) samt stänga förbindelsen igen. (Se exempel i nästa kapitel.)

5.5.7 Objektet CommandBuilder

Vi kan skapa en instans av klassen CommandBuilder för att generera värden för DataAdapters egenskaper `DeleteCommand`, `InsertCommand` och `UpdateCommand`. För att detta ska fungera så måste egenskapen `SelectCommand` har sats och `SelectCommand` får endast använda tabellnamn eller SQL-satser som läser data från endast en tabell (samt några restriktioner till ☺).

Det enda som behöver göras är att vi skapar en instans av klassen CommandBuilder och skickar vår DataAdapter som parameter till konstruktorn.

```
Dim adoDA As OleDb.OleDbDataAdapter
Dim adoCB As OleDb.OleDbCommandBuilder

'Skapa instans av CommandBuilder – skicka DataAdapter som parameter t. konstruktor
adoCB = New OleDb.OleDbCommandBuilder(adoDA)
```

5.6 Objektet DataSet

Ett DataSet-objekt innehåller data som hämtats från datakälla, d.v.s. är ett objekt som används för att lagra data temporärt i klienter samt för att transportera data mellan (data-)komponent och dess klienter. Till skillnad mot Recordset-objekt i ADO så kan DataSet-objekt innehåller flera tabeller (eller resultat från kommando) samt relationer mellan tabeller.[25] DataSet-objekt kopplas till datakälla via ett DataAdapter-objekt – ett för varje tabell som lagras i DataSet.

Klassen DataSet har definierats i namnutrymmet `System.Data` (d.v.s. inte i `System.Data.OleDb` som objekt beskrivna ovan) och är detsamma oavsett vilken typ av datakälla man använder (t.ex. SQL Server eller OLE DB).

DataSet-objekt är starkt knutna till XML, d.v.s. vi skulle med inte allt för mycket jobb kunna skicka data till andra typer av applikationer än .NET-applikationer. Det finns metoder i DataSet-objekt för att konvertera till och från XML (vilka inte beskrivs här…).

Nedan listas intressanta egenskaper och metoder i ett DataSet-objekt.

- .Relations
- .Tables
- .AcceptChanges()
- .GetChanges()
- .HasChanges()
- .RejectChanges()

5.6.1 Egenskapen Relations [UTVECKLA]

Denna egenskap är en vektor med alla relationer mellan tabellerna i aktuellt DataSet.

5.6.2 Egenskapen Tables

Denna egenskap är en vektor med alla tabeller i aktuellt DataSet, d.v.s. vår data. ”Tabellerna” kan vara hela tabeller men även resultatet från en SQL-sats eller lagrade frågor. Objekten i vektorn är av typen DataTable (se *Objektet DataTable* nedan).

[25] Recordset-objekt i ADO kunde till viss del innehålla flera tabeller, men det var inte vidare intuitivt.

För att lägga till tabeller i denna vektor använder vi metoden `Fill()` i våra DataAdapter-objekt. Och för att hämta värden från denna vektor kan vi antingen använda ett index eller det namn som vi angav när metoden `Fill()` anropades (om något).

```
Dim adoDS As Data.DataSet
Dim adoTable As Data.DataTable

'Hämta första tabellen i DataSet
adoTable = adoDS.Tables(0)
```

5.6.3 Metoden AcceptChanges [KONTROLLERA]

Denna metod återskapar förbindelse med databas och gör förändringar i DataSet beständiga i datakälla, d.v.s. utför Commit.

5.6.4 Metoden GetChanges [KONTROLLERA]

Returnerar ett DataSet-objekt med alla ändringar gjorda i aktuellt DataSet-objekt.

```
Dim adoDS As Data.DataSet, adoDS2 As Data.DataSet
Dim adoTable As Data.DataTable

'Hämta ändringar gjorda i DataSet (adoDS)
adoDS2 = adoDS.GetChanges()
```

5.6.5 Metoden HasChanges [KONTROLLERA]

Metoden returnerar sant om några ändringar (nya poster, uppdaterade fält eller raderade poster) gjorts i DataSet.

5.6.6 Metoden RejectChanges [KONTROLLERA]

Metoden förkastar alla ändringar gjorts sen DataSet fylldes (med tabeller) eller sista gången `AcceptChanges()` anropades.

5.7 Objektet DataTable

Ett DataTable-objekt motsvarar t.ex. en tabell i en databas eller resultatet från en SQL-fråga (jag kommer i beskrivningarna nedan att referera till detta objekt som ”tabellen” även om det kan vara ett resultat av en SQL-fråga). Det är även dessa objekt som lagras i ett DataSet-objekts vektor `Tables`, d.v.s. dessa objekt är främst intressanta i samband med ett DataSet-objekt men kan även skapas skilt från databas. DataTable-objekt kan användas för att skriva ut innehållet (posterna) i tabeller, men även manipulera dess data.

- Columns
- DataSet
- DefaultView
- PrimaryKey
- Rows
- AcceptChanges()
- RejectChanges()

En av de viktigaste egenskaperna i DataTable är Rows som innehåller alla poster i tabell och är ett Collection-objekt (d.v.s. en vektor).

5.7.1 Egenskapen Columns

Denna egenskap innehåller en vektor med beskrivningar av kolumner i DataTable-objektet. Beskrivningarna är objekt av typen DataColumn.

Denna egenskap är t.ex. användbar för att hämta namn på kolumner i datakälla för att skriva ut rubriker i en tabell. I exempel nedan skrivs namnet på alla kolumner ut med ett tabtecken mellan.

```
For Each adoColumn In adoTable.Columns
  Console.Write(adoColumn.ColumnName)  'Skriv ut namn på kolumn
  Console.Write(vbTab)                 'Skriv ut ett tabtecken
Next
```

För att hämta data används egenskapen `Rows` (se nedan).

5.7.2 Egenskapen DataSet

Refererar till eventuellt DataSet som DataTable tillhör.

5.7.3 Egenskapen DefaultView

Vi kan med denna egenskap t.ex. begränsa (filtrera) vilka poster i tabellen som ska visas. (Denna egenskap behandlas inte vidare i denna sammanfattning, inte nu i.a.f.. ☺)

5.7.4 Egenskapen Rows

Egenskapen `Rows` innehåller en vektor (av typen Collection) med alla poster i tabellen. Posterna är objekt av typen DataRow.

I nedanstående exempel loopas över alla poster i tabell och sedan alla kolumner i aktuell post. M.h.a. koden nedan kan alla poster och kolumner i vilken tabell som helst skrivas ut – antalet loopar (både vad gäller poster och kolumner) kan variera eftersom vi använder For Each-loop. För att kunna loopa över kolumnerna i en post använder vi egenskapen `ItemArray` för att erhålla alla kolumners värden i en vektor.

```
For Each adoRow In adoTable.Rows    'Loopa över posterna i tabell
  For Each obj In adoRow.ItemArray  'Loopa över kolumnerna i post
    Console.Write(obj)              'Skriv ut kolumnens värde
    Console.Write(vbTab)            'Skriv ut ett tabtecken
  Next
  Console.WriteLine()               'Skriv ut en radbrytning
Next
```

Även om ovanstående kod är praktisk så vill vi ofta kunna skriva ut endast delar av poster eller inte i den ordning de förekommer i tabell. Vi kan då använda en For Each-loop för att loopa över posterna och sen använda index (eller nycklar) för att hämta värden från aktuell post (den post som postpekaren pekar på nu). I exempel nedan skrivs kolumnerna 0 (första) till 2 ut m.h.a. index och sista kolumnen m.h.a. en ”nyckel”, d.v.s. namnet på kolumnen.

```
For Each adoRow In adoTable.Rows    'Loopa över posterna i tabell
  Console.Write(adoRow(0))          'Skriv ut första kolumnen i post
  Console.Write(adoRow(1))
  Console.Write(adoRow(2))
  Console.Write(adoRow("Telefon"))  'Skriv ut kolumnen med namnet Telefon
Next
```

För att hämta metadata om posterna används egenskapen `Columns` (se ovan).

5.7.5 Metoden AcceptChanges()

Denna metod fungerar som i DataSet-objektet fast bekräftar endast sparandet i aktuell tabell (d.v.s. inte alla tabeller i DataSet-objektet).

5.7.6 Metoden RejectChanges()

Denna metod fungerar som i DataSet-objektet fast rullar endast tillbaka ändringar i aktuell tabell (d.v.s. inte alla tabeller i DataSet-objektet).

5.8 Fler objekt av intresse

Det finns fler objekt (eller snarare klasser) i ADO.NET. Några av dessa är:

- DataColumn
- DataRelation
- DataRows

Några av dess används i exempel i nästa kapitel (och några kommer eventuellt att beskrivas i kommande versioner av denna sammanfattning).

6 Databasfunktioner i kod

I detta kapitel ska vi titta på mer sammanhängande kod för att få en helhet. Ibland kan exempel verka innehålla överdrivet många variabler, d.v.s. några variabler skulle kunna skippas och därmed göra koden kortare. Men alla dessa variabler finns med för att bl.a. visa typer/klasser för objekten och för att det förhoppningsvis är mer pedagogiskt att visa varje steg.

Lättaste och snabbaste sättet att hämta poster för visning (d.v.s. inte uppdatering) är att använda en DataReader som vi kan få från vårt Command-objekt. Men vi kan även använda DataSet-objekt, framför allt om vi vill skicka poster från en datakälla via komponenter i en distribuerad applikation.

Vill vi lägga till, uppdatera och ta bort poster så kan vi antingen använda ett Command-objekt (och t.ex. SQL-satser) eller ett DataSet-objekt (vilket i.o.f.s. involverar SQL-satser ☺).

I nedanstående avsnitt, om hur man öppnar en tabell och hämta poster, så visas först hur vi kan använda en DataReader och sen ett DataSet. För övriga databasfunktioner (infoga, uppdatera och ta bort) kommer exempel visas med Command-objekt och DataSet-objekt.

Variabler i exempel (i resterande avsnitt i kapitlet) använder prefixet ”ado” för att tala om att det är dataobjekt. Och i alla nedanstående exempel kommer vi använda en global (konstant) variabel i formulären som innehåller vår ConnectionString (ändra eventuellt sökvägen till databasfil).

```
Private Const cstrConn As String = "Provider=Microsoft.Jet.OLEDB.4.0;" _
                        & "Data Source=C:\bokning.mdb;Persist Security Info=False"
```

Koden i exempel nedan kan placeras i metoden `Form_Load()` eller i en händelsehanterare för knappar.

6.1 Öppna tabell och hämta poster

Lättaste och snabbaste sättet att hämta poster för visning (d.v.s. inte uppdatering) är, som sagt, att använda en DataReader som vi kan få från vårt Command-objekt. Därför börjar vi med att hämta poster med personal samt skriva ut signatur, förnamn och efternamn i en listruta.

6.1.1 Skapa ett Connection-objekt implicit

Även i ADO.NET kan vi skapa Connection-objekt implicit, d.v.s. genom att bara skicka en ConnectionString istället för ett Connection-objekt. Med ADO.NET ska detta inte vara ett problem då alla *data providers* bör implementera en förbindelsepool (*connection pool*). Det **viktiga** är dock att vi använder **identiska** ConnectionString för att poolningen ska fungera. Lämpligen deklareras dessa ConnectionString som konstanter (som i dessa exempel ☺).

6.1.2 Öppna tabell och hämta poster med DataReader

Nedan beskrivs hur man använder en DataReader för att lista alla poster i en tabell i ett formulärs listruta.

Först måste vi skapa ett Connection-objekt för förbindelse till datakälla samt öppna förbindelsen till datakällan. Därefter skapas ett Command-objekt där vi anger kommando att utföra (t.ex. en SQL-fråga), typ av kommando samt vilken dataförbindelse att använda. Sen anropar vi metoden `ExecuteReader()`, i Command-objektet för att utföra kommando och

returnerar ett DataReader-objekt. I detta exempel hämtas data som placeras i en sträng (av typen StringBuilder[26] då vi sammanfogar strängar) för att sen läggas till i en listruta. Sist, och mycket viktigt, så stängs DataReader.

```
Dim adoConn As OleDb.OleDbConnection, adoCmd As OleDb.OleDbCommand
Dim adoReader As OleDb.OleDbDataReader
Dim strSQL As String, strTemp As System.Text.StringBuilder

strSQL = "SELECT * FROM tblPersonal"

  'Skapa Connection-objekt och öppna datakälla
adoConn = New OleDb.OleDbConnection(cstrConn)
adoConn.Open()

  'Skapa Command-objekt och sätt egenskaper för objektet
adoCmd = New OleDb.OleDbCommand()
adoCmd.CommandText = strSQL              'SQL-sats att utföra
adoCmd.CommandType = CommandType.Text    'Typ av kommand (SQL-sats)
adoCmd.Connection = adoConn              'DB-förbindelse att använda

  'Kör fråga och placera resultatet i adoReader - stäng Conn.-objekt med Reader
adoReader = adoCmd.ExecuteReader(CommandBehavior.CloseConnetion)

  'Så länge det finns fler poster - lägg till person i listruta
While adoReader.Read
    'Skapa sträng med data från aktuell post
  strTemp = New System.Text.StringBuilder(adoReader.GetString("Id"))
  strTemp.append(" ")
  strTemp.append(adoReader.GetString("Fnamn"))
  strTemp.append(" ")
  strTemp.append(adoReader.GetString("Enamn"))

  ListBox1.Items.Add(strTemp)           'Lägg till sträng i listruta
End While

adoReader.Close()                       'Stäng DataReader och Connection-objekt
```

6.1.3 Öppna tabell och hämta poster med DataSet

Nedan beskrivs hur man använder ett DataSet för att lista alla poster i en tabell i ett formulärs listruta.

Precis som med DataReader så skapar vi ett Connection-objekt (och öppnar) samt ett Command-objekt (utan att utföra kommando). Här visas också hur vi kan använda konstruktor för att sätta egenskaper för Command-objekt. Därefter skapar vi en DataAdapter och bifogar Command-objektet till dess konstruktor. Sist av allt skapar vi ett tomt DataSet som vi kopplar samman med DataAdapter genom att anropa metoden `Fill()`.

Eftersom de flesta vektorerna i ADO.NET är av typen Collection så kan vi loopa över samlingar med For Each-loopar. För att loopa över posterna i en tabell måste vi först hämta posterna från tabellen. Detta gör vi genom att fråga vårt DataSet efter en tabell och sen genom att fråga tabellen om dess poster (`adoDS.Tables("tblPersonal").Rows`).

```
Dim adoConn As OleDb.OleDbConnection, adoCmd As OleDb.OleDbCommand
Dim adoDA As OleDb.OleDbDataAdapter, adoDS As Data.DataSet
Dim adoRow As Data.DataRow
Dim strSQL As String, strTemp As System.Text.StringBuilder

strSQL = "SELECT * FROM tblPersonal"

  'Skapa Connection-objekt och öppna förbindelse till datakälla
```

[26] Eftersom namnutrymmet System.Text inte importeras så används det fullständiga namnet på klassen, d.v.s. på formen Namnutrymme.Klassnamn.

```
adoConn = New OleDb.OleDbConnection(cstrConn)
adoConn.Open()

  'Skapa Command-, DataAdapter- och DataSet-objekt
adoCmd = New OleDb.OleDbCommand(strSQL, adoConn)
adoCmd.CommandType = CommandType.Text          'Ange typ av kommando
adoDA = New OleDb.OleDbDataAdapter(adoCmd)
adoDS = New Data.DataSet()

  'Fyll DataSet med data och namnge tabell
adoDA.Fill(adoDS, "tblPersonal")

  'Loopa över alla rader (poster) i DataSets tabell tblPersonal
For Each adoRow In adoDS.Tables("tblPersonal").Rows
  strTemp = New System.Text.StringBuilder()  'Skapa StringBuilder-objekt
  strTemp.Append(adoRow.Item("Id"))          'Lägg till data i sträng
  strTemp.Append(" ")
  strTemp.Append(adoRow.Item("FNamn"))
  strTemp.Append(" ")
  strTemp.Append(adoRow.Item("ENamn"))

  ListBox1.Items.Add(strTemp.ToString())     'Lägg till sträng i listruta
Next

adoConn.Close()                              'Stäng förbindelse till datakälla
```

I exempel ovan använder vi ett StringBuilder-objekt för att skapa strängen som ska skrivas ut i listruta. När vi lägger till sträng i listruta används metoden `ToString()` för att erhålla den sträng som vårt StringBuilder-objekt innehåller.

6.2 Lägga till post i tabell

För att lägga till poster i en databas kan vi antingen använda ett DataSet-objekt (i kombination med ett DataAdapter-objekt) eller en SQL-sats och ett Command-objekt.

6.2.1 Lägga till en post i tabell med Command-objekt

Precis som när vi läste poster i en tabell så måste vi ha en förbindelse med databasen, d.v.s. ett Connection-objekt. Därefter skapar vi ett Command-objekt och anger egenskapen `CommandText` samt anropar metoden `ExecuteNonQuery()`. Vi använder denna metod eftersom vår SQL-sats kommer vara av typen INSERT som inte returnerar några poster.

Vi börjar med att deklarera variabler, bl.a. för våra dataobjekt. Den stora skillnaden mellan nedanstående exempel och det med DataReader ovan är att SQL-satsen är av typen INSERT samt att den innehåller frågetecken där vi vill infoga variabla värden (d.v.s. själva värdena) i SQL-satsen. För varje frågetecken i SQL-satsen måste vi sen skapa ett Parameter-objekt och ange värden för dess egenskaper, främst då `Value`. Parameter-objekten lägger vi sen till i Command-objektets vektor `Parameters`.

```
Dim adoConn As OleDb.OleDbConnection, adoCmd As OleDb.OleDbCommand
Dim adoParam As OleDb.OleDbParameter
Dim strConn, strSQL, strSignatur, strFnamn, strEnamn, strTelefon As String

  'Här borde vi hämta värden från t.ex. textrutor i formulär - men här används
  ' konstanta värden för exempel
strSignatur = "een"
strFnamn = "Erik"
strEnamn = "Eriksson"
strTelefon = "016-11 22 33"

  'Skapa SQL-sats med parametrar
strSQL = "INSERT INTO tblPersonal(Id, FNamn, ENamn, Telefon) VALUES(?, ?, ?, ?)"

  'Skapa Connection-objekt och öppna förbindelse med datakälla
```

```
adoConn = New OleDb.OleDbConnection(cstrConn)
adoConn.Open()

  'Skapa Command-objekt
adoCmd = New OleDb.OleDbCommand()
adoCmd.CommandText = strSQL              'SQL-sats att utföra
adoCmd.CommandType = CommandType.Text    'Typ av kommand (SQL-sats)
adoCmd.Connection = adoConn              'DB-förbindelse att använda

  'Skapa Parameter-objekt (ett för varje parameter i SQL-sats) och ange värden
adoParam = adoCmd.CreateParameter()
adoParam.Value = strSignatur
adoParam.Direction = ParameterDirection.Input
adoCmd.Parameters.Add(adoParam)                  'Lägg till i Command-objekt

adoParam = adoCmd.CreateParameter()
adoParam.Value = strFnamn
adoParam.Direction = ParameterDirection.Input
adoCmd.Parameters.Add(adoParam)

adoParam = adoCmd.CreateParameter()
adoParam.Value = strEnamn
adoParam.Direction = ParameterDirection.Input
adoCmd.Parameters.Add(adoParam)

adoParam = adoCmd.CreateParameter()
adoParam.Value = strTelefon
adoParam.Direction = ParameterDirection.Input
adoCmd.Parameters.Add(adoParam)

adoCmd.ExecuteNonQuery()                         'Utför kommando
```

Vi kan återanvända variabeln `adoParam` då objekten vi skapar lagras i vektorn `Parameters`, d.v.s. det finns ingen anledning att skapa en variabel för varje parameter.

6.2.2 Lägga till en post i tabell med DataSet-objekt

Att lägga till poster med DataAdapter-objekt fungerar på ett liknande sätt som när vi endast läser posterna från tabellen. Men vi måste ange ett värde för egenskapen `InsertCommand` i vårt DataSet-objekt. Vill vi inte ange värdet själv så kan vi använda en ”hjälpklass” (CommandBuilder) för att generera värden för de tre egenskaperna `Insert-`, `Update-` och `DeleteCommand`. Hjälpklassen gör detta utifrån värdet på egenskapen `SelectCommand`, d.v.s. värdet för denna egenskap måste anges först (och en del andra saker uppfyllas[27]).

När vi fyllt vårt DataSet-objekt så måste vi hämta tabellen, som vi ska lägga till en post i, från DataSet-objektet. Sen anropar vi metoden `NewRow()` i tabellobjektet för att skapa en ny post, fyller posten med data samt lägger till posten i tabellens vektor `Rows` (igen ☺). `NewRow()` returnerar ett objekt av typen DataRow som innehåller en vektor `Item` med en position för varje kolumn i tabellobjektet. Sist uppdaterar vi datakällan genom att anropa metoden `Update()` i DataAdapter-objektet.

Om vi lagrar vårt DataSet-objekt i en global variabel i klienten (som i exempel sist i detta kapitel) så måste vi även anropa metoden `AcceptChanges()` i DataSet-objektet.

```
Dim adoConn As OleDb.OleDbConnection, adoCmd As OleDb.OleDbCommand
Dim adoDA As OleDb.OleDbDataAdapter, adoCB As OleDb.OleDbCommandBuilder
Dim adoDS As Data.DataSet, adoTable As Data.DataTable
Dim adoRow As Data.DataRow
Dim strSQL, strTemp, strSignatur, strFnamn, strEnamn, strTelefon As String
```

[27] T.ex. så måste SelectCommand för DataAdapter vara en SELECT-sats som hämtar från endast en tabell och alla kolumner som har restriktionen NOT NULL.

```
    'Här borde vi hämta värden från t.ex. textrutor i formulär - men här
    ' används konstanta värden för exempel
  strSignatur = "bbn"
  strFnamn = "Bengt"
  strEnamn = "Bengtsson"
  strTelefon = "3678"

  strSQL = "SELECT * FROM tblPersonal"

    'Skapa Connection-objekt och öppna förbindelse till datakälla
  adoConn = New OleDb.OleDbConnection(cstrConn)
  adoConn.Open()

    'Skapa Command-, DataAdapter-, CommandBuilder- och DataSet-objekt
  adoCmd = New OleDb.OleDbCommand(strSQL, adoConn)
  adoDA = New OleDb.OleDbDataAdapter(adoCmd)
  adoCB = New OleDb.OleDbCommandBuilder(adoDA)
  adoDS = New Data.DataSet()

    'Fyll DataSet med data och namnge tabell
  adoDA.Fill(adoDS, "tblPersonal")

    'Hämta tabell att lägga till i
  adoTable = adoDS.Tables("tblPersonal")

    'Skapa en ny rad (post) från tabell
  adoRow = adoTable.NewRow

    'Sätt värden på fält i rad
  adoRow.Item("Id") = strSignatur
  adoRow.Item("Fnamn") = strFnamn
  adoRow.Item("Enamn") = strEnamn
  adoRow.Item("Telefon") = strTelefon

    'Lägg till rad i tabell
  adoTable.Rows.Add(adoRow)

    'Uppdatera datakälla som DataSets tabell finns i
  adoDA.Update(adoDS, "tblPersonal")

  adoConn.Close()                    'Stäng förbindelse till datakälla
```

Observera att här (och i exempel nedan med uppdatering och radering av poster) hämtas alla poster i tabellen innan ny post infogas (respektive uppdateras/raderas). Detta är inte vidare praktiskt om tabellen innehåller tusentals med poster och vi endast ska lägga till en post. Mer praktiskt är m.a.o. att använda ett Command-objekt och en INSERT-sats. Att använda ett DataSet-objekt är främst praktiskt om vi t.ex. använder ett Windows-gränssnitt där vi presenterar alla poster i ett formulär för användaren att ändra/lägga till poster.

6.3 Uppdatera post i tabell

Även för att uppdatera poster i en databas så kan vi antingen använda ett DataSet-objekt (i kombination med ett DataAdapter-objekt) eller ett Command-objekt.

6.3.1 Uppdatera en post i tabell med Command-objekt

Om vi använder ett Command-objekt så använder vi en SQL-sats av typen UPDATE, vilket fungerar på ett liknande sätt som när vi lade till en post i tabellen (d.v.s. koden är nästan den samma). Observera ordning på parametrar!

```
  Dim adoConn As OleDb.OleDbConnection, adoCmd As OleDb.OleDbCommand
  Dim adoParam As OleDb.OleDbParameter
  Dim strSQL, strSignatur, strFnamn, strEnamn, strTelefon As String

    'Här borde vi hämta värden från t.ex. textrutor i formulär - men här
    ' används konstanta värden för exempel
```

```
strSignatur = "aan"
strFnamn = "Adam"
strEnamn = "Adamsson"
strTelefon = "3679"

  '"Bygg" SQL-sats med parametrar (frågetecken - ?)
strSQL = "UPDATE tblPersonal SET Fnamn=?, Enamn=?, Telefon=? WHERE Id=?"

  'Skapa Connection-objekt och öppna förbindelse till datakälla
adoConn = New OleDb.OleDbConnection(cstrConn)
adoConn.Open()

  'Skapa Command-, DataAdapter- och DataSet-objekt
adoCmd = New OleDb.OleDbCommand(strSQL, adoConn)

  'Skapa Parameter-objekt (ett för varje parameter i SQL-sats) och ange värden
adoParam = adoCmd.CreateParameter()
adoParam.Value = strFnamn
adoCmd.Parameters.Add(adoParam)     'Lägg till i Command-objekt

adoParam = adoCmd.CreateParameter()
adoParam.Value = strEnamn
adoCmd.Parameters.Add(adoParam)

adoParam = adoCmd.CreateParameter()
adoParam.Value = strTelefon
adoCmd.Parameters.Add(adoParam)

adoParam = adoCmd.CreateParameter()
adoParam.Value = strSignatur
adoCmd.Parameters.Add(adoParam)

Try
  adoCmd.ExecuteNonQuery()          'Utför kommando
Catch ex As Exception
  MessageBox.Show(ex.Message)       'Visa meddelanderuta med eventuellt fel
End Try

adoConn.Close()                     'Stäng förbindelse till datakälla
```

6.3.2 Uppdatera en post i tabell med DataSet-objekt

På motsvarande sätt som när vi lägger till en ny post så måste vi ange ett värde för egenskapen `UpdateCommand` i vårt DataAdapter-objekt innan vi kan uppdatera data med DataSet-objekt. Återigen kan vi använda ett CommandBuilder-objekt för detta.

För att uppdatera en post (bland många andra) så måste vi först hitta posten. Ett sätt att söka fram en viss post är baserat på primärnyckel i tabell. Men för att kunna söka på primärnyckel i ett DataSet så måste vi först ange primärnyckel för tabell. Vi måste alltså hämta kolumn(er) som är primärnyckel, vilka placeras i en vektor (*array*) av typen DataColumn, samt använda egenskapen `PrimaryKey` i DataTable-objekt.

Därefter kan vi använda metoden `Find()` i vårt DataTable-objekt, vilken returnerar ett DataRow-objekt (primärnyckeln är unik, så endast en post kan returneras). Om metoden returnerade ett objekt så har vi hittat vår post och kan uppdatera dess data.

Övrig kod är lik den som används för att infoga en post i en tabell.

```
Dim adoConn As OleDb.OleDbConnection, Dim adoCmd As OleDb.OleDbCommand
Dim adoDA As OleDb.OleDbDataAdapter, Dim adoCB As OleDb.OleDbCommandBuilder
Dim adoDS As Data.DataSet, adoTable As Data.DataTable
Dim adoRow As Data.DataRow
Dim strSQL, strTemp, strSignatur, strFnamn, strEnamn, strTelefon As String
Dim arrKey(0) As DataColumn  'För PK - PK enkel (ej sammansatt) - en pos i vektor

strSignatur = "bbn"
strFnamn = "Benny"
strEnamn = "Benktsson"
strTelefon = "3678"
```

```
strSQL = "SELECT * FROM tblPersonal"

  'Skapa Connection-objekt och öppna förbindelse till datakälla
adoConn = New OleDb.OleDbConnection(cstrConn)
adoConn.Open()

  'Skapa Command-, DataAdapter- och DataSet-objekt
adoCmd = New OleDb.OleDbCommand(strSQL, adoConn)
adoDA = New OleDb.OleDbDataAdapter(adoCmd)
adoCB = New OleDb.OleDbCommandBuilder(adoDA)
adoDS = New Data.DataSet()

  'Fyll DataSet med data och namnge tabell
adoDA.Fill(adoDS, "tblPersonal")

  'Hämta tabell att söka i
adoTable = adoDS.Tables("tblPersonal")

  'Hämta kolumn som är PK (i datakälla) och sätt PK för tabellobjekt
arrKey(0) = adoTable.Columns("Id") 'Hämta PK för denna tabell
adoTable.PrimaryKey = arrKey       'Ange PK för tabellobjekt

  'Sök fram post baserat på värde på PK
adoRow = adoTable.Rows.Find(strSignatur)

  'Om post hittades - uppdatera data...
If Not adoRow Is Nothing Then
  adoRow.Item("Fnamn") = strFnamn
  adoRow.Item("Enamn") = strEnamn
  adoRow.Item("Telefon") = strTelefon
  adoDA.Update(adoDS, "tblPersonal")  'Uppdatera datakälla
  adoDS.AcceptChanges()               'Uppdatera DataSet
Else
  MessageBox.Show("Hittade inte post...")
End If

adoConn.Close()                     'Stäng förbindelse till datakälla
```

6.4 Ta bort post i tabell

Även för att ta bort poster i en databas så kan vi antingen använda ett DataSet-objekt (i kombination med ett DataAdapter-objekt) eller ett Command-objekt.

6.4.1 Ta bort en post i tabell med Command-objekt

Om vi använder ett Command-objekt så använder vi en SQL-sats av typen DELETE, vilket fungerar på ett liknande sätt som när vi lade till en post i tabellen (d.v.s. koden är nästan den samma). Här behöver vi endast parameter för primärnyckeln (en i detta exempel).

```
Dim adoConn As OleDb.OleDbConnection, adoCmd As OleDb.OleDbCommand
Dim adoParam As OleDb.OleDbParameter
Dim strSQL, strSignatur As String

  'Här borde vi hämta värden från t.ex. textrutor i formulär - men här
  ' används konstanta värden för exempel
strSignatur = "aan"

  '"Bygg" SQL-sats med parametrar (frågetecken - ?)
strSQL = "DELETE FROM tblPersonal WHERE Id=?"

  'Skapa Connection-objekt och öppna förbindelse till datakälla
adoConn = New OleDb.OleDbConnection(cstrConn)
adoConn.Open()

  'Skapa Command-, DataAdapter- och DataSet-objekt
adoCmd = New OleDb.OleDbCommand(strSQL, adoConn)

  'Skapa Parameter-objekt (ett för varje parameter i SQL-sats) och ange värden
adoParam = adoCmd.CreateParameter()
adoParam.Value = strSignatur
```

```
adoCmd.Parameters.Add(adoParam)      'Lägg till i Command-objekt

Try
  adoCmd.ExecuteNonQuery()           'Utför kommando
Catch ex As Exception
  MessageBox.Show(ex.Message)
End Try

adoConn.Close()                      'Stäng förbindelse till datakälla
```

6.4.2 Ta bort en post i tabell med DataSet-objekt

På motsvarande sätt som när vi lägger till en ny post så måste vi ange ett värde för egenskapen `DeleteCommand` i vårt DataAdapter-objekt (återigen kan vi använda CommandBuilder ☺). Även här måste vi finna posten innan vi kan ta bort den.

När vi funnit posten anropar vi metoden `Delete()` i postobjektet och uppdaterar datakälla med DataAdapter och sen DateSet-objektet med metoden `AcceptChanges()`.

```
Dim adoConn As OleDb.OleDbConnection, adoCmd As OleDb.OleDbCommand
Dim adoDA As OleDb.OleDbDataAdapter, adoCB As OleDb.OleDbCommandBuilder
Dim adoDS As Data.DataSet, adoTable As Data.DataTable
Dim adoRow As Data.DataRow, strSQL As String
Dim strSignatur, strFnamn, strEnamn, strTelefon As String
Dim arrKey(0) As DataColumn

strSignatur = "bbn"
strFnamn = "Benny"
strEnamn = "Benktsson"
strTelefon = "3678"

strSQL = "SELECT * FROM tblPersonal"

'Skapa Connection-objekt och öppna förbindelse till datakälla
adoConn = New OleDb.OleDbConnection(cstrConn)
adoConn.Open()

'Skapa Command-, DataAdapter- och DataSet-objekt
adoCmd = New OleDb.OleDbCommand(strSQL, adoConn)
adoDA = New OleDb.OleDbDataAdapter(adoCmd)
adoCB = New OleDb.OleDbCommandBuilder(adoDA)
adoDS = New Data.DataSet()

'Fyll DataSet med data och namnge tabell
adoDA.Fill(adoDS, "tblPersonal")

'Hämta tabell att söka i
adoTable = adoDS.Tables("tblPersonal")

'Hämta kolumn som är PK och sätt PK för tabell
arrKey(0) = adoTable.Columns(0) 'Hämta första kolumn - PK för denna tabell
adoTable.PrimaryKey = arrKey    'Ange PK för tabellobjekt

'Sök fram post baserat på värde på PK
adoRow = adoTable.Rows.Find(strSignatur)

'Om post hittades - uppdatera data...
If Not adoRow Is Nothing Then
  adoRow.Delete()
  adoDA.Update(adoDS, "tblPersonal")
  adoDS.AcceptChanges()
Else
  MessageBox.Show("Hittade inte post...")
End If

adoConn.Close()                      'Stäng förbindelse till datakälla
```

6.5 Skapa obundna DataSet-objekt

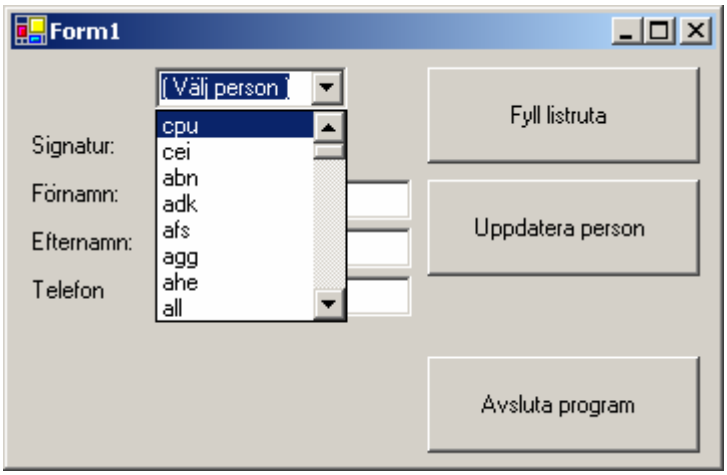
Detta är ett större exempel för att bl.a. visa på hur obundna DataSet-objekt kan användas och hur vi ansluter till datakälla igen för uppdateringar. Exemplet bör **inte** ses som typiskt exempel för enanvändarsystem bör fungera – i enanvändarsystem bör förbindelse till datakälla öppnas när applikation startas och hållas öppen tills applikation avslutas. Men i fleranvändarsystem så bör vi inte hålla tag i databasanslutningar längre än vi behöver.

DataSet-objekt är som standard obundna, d.v.s. tänkta att kunna kopplas loss från datakälla. Vi använder DataAdapter-objekt för att hantera eventuella uppdateringar mot datakällan. Anslutning till datakälla bör stängas när vi fyllt våra DataSet samt förbindelse etableras igen för att utföra uppdateringar av datakälla.

I nedanstående exempel visas hur vi öppnar datakälla, fyller ett DataSet-objekt med data (från personaltabellen) samt stänger datakälla igen när användaren klickar på knappen Fyll listruta. DataSet-objektet sparas i en ”global” variabel (d.v.s. en instansvariabel `madoDS`) för att vara tillgänglig i andra händelsehanterare (metoder). När användaren väljer en signatur i komboboxen så hämtas data från DataSet och kopieras till formulärets textrutor. Om användaren ändrar data i textrutorna kan användaren klicka på knappen Uppdatera person för att spara ändringar i datakälla. För att spara ändringar i datakälla måste vi först ändra data i vårt DataSet-objekt, etablera kontakt med datakälla (via DataAdapter-objekt), uppdatera och stänga förbindelse igen.

Kontrollerna i formuläret heter enligt följande:

- komboboxen heter `lstPersonal`.
- de fyra textrutorna (bakom komboboxens lista i bild nedan) `txtSignatur`, `txtFNamn`, `txtENamn` och `txtTelefon`.
- samt knapparna `btnFyll`, `btnUppdatera` och `btnAvsluta`.



Figur 5 - Formulär för exempel med obundet DataSet-objekt.

Vi börjar med att deklarera konstanter för ConnectionString och SQL-sats för att hämta data från tabell samt en instansvariabel för att hålla vårt obundna DataSet-objekt.

```
'*** Konstanter ***
'ConnectionString (ändra eventuellt sökväg till databasfil)
Private Const cstrConn As String = "Provider=Microsoft.Jet.OLEDB.4.0;" _
  & "Data Source=C:\bokning.mdb;Persist Security Info=False"
```

```
'SQL-sats
Private Const cstrSQL As String = "SELECT * FROM tblPersonal"

'*** Instansvariabler ***
Private madoDS As Data.DataSet
```

Därefter skriver vi koden för händelsehanterare bakom knappen Fyll listruta.[28] Här etableras kontakt med datakälla, DataSet-objekt skapas och fylls, listruta fylls med data från DataSet-objekt samt förbindelse med datakälla stängs. Eftersom vi ska söka efter poster (m.h.a. primärnyckel) så måste vi även ange primärnyckel för tabell. Detta görs lämpligen här då uppdateringar kan ske flera gånger, men kod här mer sällan.

```
Private Sub btnFyll_Click(ByVal sender As System.Object, _
                         ByVal e As System.EventArgs) Handles btnFyll.Click
  Dim adoConn As OleDb.OleDbConnection, adoCmd As OleDb.OleDbCommand
  Dim adoDA As OleDb.OleDbDataAdapter, adoCB As OleDb.OleDbCommandBuilder
  Dim adoTable As Data.DataTable, adoRow As Data.DataRow
  Dim arrKey(0) As DataColumn

  'Skapa Connection-objekt och öppna förbindelse till datakälla
  adoConn = New OleDb.OleDbConnection(cstrConn)
  adoConn.Open()

  'Skapa Command-, DataAdapter- och DataSet-objekt
  adoCmd = New OleDb.OleDbCommand(cstrSQL, adoConn)
  adoDA = New OleDb.OleDbDataAdapter(adoCmd)
  adoCB = New OleDb.OleDbCommandBuilder(adoDA)
  madoDS = New Data.DataSet()

  'Fyll DataSet med data och namnge tabell
  adoDA.Fill(madoDS, "tblPersonal")

  'Hämta tabell att ange PK för
  adoTable = madoDS.Tables("tblPersonal")

  'Hämta kolumn som är PK och sätt PK för tabell (för sökning)
  arrKey(0) = adoTable.Columns("Id") 'Hämta Id-kolumn - PK för denna tabell
  adoTable.PrimaryKey = arrKey       'Ange PK för tabellobjekt

  lstPersonal.Items.Clear()   'Töm listruta på värden

  'Fyll listruta med värden från DataSet
  For Each adoRow In adoTable.Rows
    lstPersonal.Items.Add(adoRow.Item("id"))
  Next

  adoConn.Close()                    'Stäng förbindelse till datakälla
End Sub
```

Eftersom vi vill att data från vårt DataSet-objekt ska visas i formulärets textrutor när användaren väljer (markerar) ett namn i komboboxen så dubbelklickar vi på komboboxen för att erhålla signaturen för händelsehanteraren nedan. Fyll sedan i koden nedan.

Observera att vi använder vårt DataSet-objekt här och att vi inte ansluter till någon datakälla.

```
Private Sub lstPersonal_SelectedIndexChanged(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles lstPersonal.SelectedIndexChanged
  Dim adoTable As Data.DataTable, adoRow As Data.DataRow
  Dim strSignatur As String
```

[28] Denna kod bör kanske ligga i formulärets metod Load() – men koden har lagts ”bakom” en knapp för att den ska kunna köras fler gånger än en (om så önskas ☺).

```
    strSignatur = lstPersonal.SelectedItem      'Hämta markerat objekt i kombobox

    'Hämta tabell att söka i
    adoTable = madoDS.Tables("tblPersonal")

    'Sök fram post baserat på värde på PK
    adoRow = adoTable.Rows.Find(strSignatur)    'Hämta post med markerad signatur

    'Om post hittades - fyll textrutor med data från DataSet...
    If Not adoRow Is Nothing Then
      txtId.Text = adoRow.Item("Id")
      txtFNamn.Text = adoRow.Item("Fnamn")
      txtENamn.Text = adoRow.Item("Enamn")

      If Not IsDBNull(adoRow.Item("Telefon")) Then  'Om telefon inte är NULL i DB...
        txtTelefon.Text = adoRow.Item("Telefon")    '... fyll textruta med data
      Else
        txtTelefon.Text = ""                        '... annars en tom sträng
      End If
    End If
  End Sub
```

För att uppdatera data om en person så börjar vi med att hämta data från textrutor i formulär.

Nästa steg är att hitta post som ska uppdateras – detta görs genom att använda primärnyckeln (PK) i vår tabell. Om vi hittade en post så uppdaterar vi kolumnerna i posten, etablerar kontakt med datakälla, uppdaterar data (m.h.a. DataAdapter-objekt) och DataSet-objekt samt stänger förbindelse till datakälla. Vi etablerar kontakt med datakälla först när vi vet att vi ska uppdatera något.

```
  Private Sub btnUppdatera_Click(ByVal sender As System.Object, _
                           ByVal e As System.EventArgs) Handles btnUppdatera.Click
    Dim adoConn As OleDb.OleDbConnection, adoCmd As OleDb.OleDbCommand
    Dim adoDA As OleDb.OleDbDataAdapter, adoCB As OleDb.OleDbCommandBuilder
    Dim adoTable As Data.DataTable, adoRow As Data.DataRow
    Dim strSignatur, strFnamn, strEnamn, strTelefon As String

    'Hämta data från textrutor
    strSignatur = txtId.Text
    strFnamn = txtFNamn.Text
    strEnamn = txtENamn.Text
    strTelefon = txtTelefon.Text

    'Hämta tabell att söka i
    adoTable = madoDS.Tables("tblPersonal")

    'Sök fram post baserat på värde på PK
    adoRow = adoTable.Rows.Find(strSignatur)

    'Om post hittades - uppdatera data...
    If Not adoRow Is Nothing Then
      adoRow.Item("Fnamn") = strFnamn     'Uppdatera kolumner i hittad post
      adoRow.Item("Enamn") = strEnamn
      adoRow.Item("Telefon") = strTelefon

      'Skapa Connection-objekt och öppna förbindelse till datakälla
      adoConn = New OleDb.OleDbConnection(cstrConn)
      adoConn.Open()

      'Skapa Command-, DataAdapter- och CommandBuilder-objekt
      adoCmd = New OleDb.OleDbCommand(cstrSQL, adoConn)
      adoDA = New OleDb.OleDbDataAdapter(adoCmd)
      adoCB = New OleDb.OleDbCommandBuilder(adoDA)

      adoDA.Update(madoDS, "tblPersonal")  'Uppdatera datakälla
      madoDS.AcceptChanges()               'Acceptera ändringar i DataSet

      adoConn.Close()                     'Stäng förbindelse till datakälla

      MessageBox.Show("Uppdaterade post...")
```

```
    Else
      MessageBox.Show("Hittade inte post...")
    End If
  End Sub
```

Vill vi även kunna stänga programmet med en knapp så fyller vi i nedanstående kod för Avsluta-knappens händelsehanterare.

```
  Private Sub btnAvsluta_Click(ByVal sender As System.Object, _
                               ByVal e As System.EventArgs) Handles btnAvsluta.Click
    Me.Close()
  End Sub
```

6.5.1 Vad som kan ändras…

Exempel ovan är inte det mest exemplariska om man ser till dess helhet… T.ex. så innehåller de flesta metoderna kod för att öppna förbindelse med databas, något som skulle kunna flyttas till en metod för att inte upprepas.

Vi skulle även kunna använda databasgränssnitten i namnutrymmet System.Data som typer på variabler. Och istället för att skapa många av objekten med konstruktorer så kan vi använda metoder som `CreateCommand()` i Connection-objekt för att skapa många av objekten. På detta sätt så blir det lättare att byta *data provider* (och därmed databas).

7 Litteratur och webbadresser

7.1 Litteratur

Denna beskrivning har utgått från sammanfattningarna *Visual Basic 6 för komponenter* och nedan finns annan litteratur som använts som referensmaterial.

- Anderson, R. et al, *Professional ASP.NET*, Wrox Press, 2001.
- Barwell, F. et al, *Professional VB.NET*, Wrox Press, 2001.
- Hoffman, K., et al, *Professional .NET Framework*, Wrox Press, 2001.
- Homer, A. & D. Sussman, *ASP.NET Distributed Data Applications*, Wrox Press, 2002.
- Löwy, J., *COM and .NET Component Services*, O'Reilly, 2001.
- MacClure, W.B. & J.J. Croft IV, *Building Highly Scalable Database Applications with .NET*, Wiley Publishing, 2002.
- Robinson, Ed, et al, *Upgrading MS Visual Basic 6.0 to MS Visual Basic.NET*, Microsoft Press, 2002.
- Robinson, S, et al, *Professional C#*, Wrox Press, 2001.
- Roman, S., et al, *VB.NET Language in a Nutshell*, O'Reilly, 2001.
- Thai, T. & H.Q. Lam, *.NET Framework Essentials*, O'Reilly, 2002.

samt webbsidor hos Microsoft (bl.a. MSDN) och webbplatsen GotDotNet.com.

7.2 Webbadresser

- `www.microsoft.com/net/` – .NET hos Microsoft.
- `www.sharpdevelop.net` – skapare av SharpDevelop.
- `www.mono.org` – skapare av GNU-versionen av .NET Framework för bl.a. Linux samt utvecklingsmiljön MonoDevelop.