

MÄLARDALENS HÖGSKOLA
Institutionen för ekonomi och informatik

Visual Basic 6 för komponenter

EI0230 – Komponentbaserad applikationsutveckling

Björn Persson, oktober 2003

Om detta sammanfattning


Avsikten med denna sammanfattning är att ge en något djupare kunskap om Visual Basic 6 som krävs för programmering av komponent med Microsofts teknologier. För att få ut mest av denna sammanfattning krävs en grundläggande förståelse för Visual Basic, databaser och objektorienterad programmering. Observera att detta **inte är en ersättning för eventuell kurslitteratur**.

Första kapitlet repeterar och utökar förståelsen av variabler, objekt, klasser, vektorer av klassen Collection och andra detaljer som vanligen inte behandlats i grundkurser i programmering med Visual Basic. En introduktion till objektorienterad programmering med Visual Basic tas upp i kapitel två. Tredje kapitlet fortsätter med databasprogrammering med *ActiveX Data Objects* (ADO). Exempel i kapitel fyra bygger på databasprogrammeringen i komponenter, d.v.s. utan datakontroller som används i användargränssnitt. I femte kapitlet behandlas felhantering. Sist behandlas övrigt.

Koden i denna sammanfattning har skrivits i Microsofts *Visual Studio 6 Professional* (Service Pack 4 och 5) och sedan kopierats in i dokumentet. Fel kan dock ha införts av misstag vid redigering av dokumentet. Övrig programvara från Microsoft som använts är *Windows NT4/2000/XP*, *Access 2000/XP* och *Visio 2002* (för diagram). För delar av koden har även *Oracle 8/9* använts.

Tabell i Access

I denna sammanfattning kommer en tabell med namnet `tblPersonal` att användas för exempel. Tabellen har skapats i en Access-databas med namnet `BOKNING.MDB` och har följande design:

Fältnamn	Datatyp	Beskrivning
 ID	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)

Databaskoden bör dock fungera även med andra databaser.

Konventioner i sammanfattning

I sammanfattning används ordet **metod** (för att använda ett gemensamt namn) för att referera till Visual Basics procedurer (Sub) och funktioner (Function). Om det i texten avses en procedur eller funktion så kommer orden **procedur** resp. **funktion** att användas.

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.

Vissa engelska begrepp saknar 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*).

Kod har skrivits med typsnitt av fast teckenbredd (Courier New) för att göras mer lätläst samt längre exempel har inneslutits i en ram (se exempel nedan).

I Visual Basic kan programsatser (*statements*) numera skrivas på flera rader genom att använda understrykningstecknet ("_"). Detta underlättar kodande och läsning av kod då man slipper skrolla i sidled för att läsa 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, oktober 2003

Björn Persson, e-post: bjorn.persson@mdh.se
Mälardalens högskola
Institutionen för ekonomi och informatik
<http://www.eki.mdh.se/personal/bpn01/>

Innehållsförteckning

1	DEKLARERA VARIABLER OCH GRUNDLÄGGANDE STRUKTURER	5
1.1	Grundläggande datatyper i Visual Basic	5
1.1.1	Deklarera variabler.....	5
1.1.2	Variabler och tilldelning av enkla värden	5
1.2	Objekt i Visual Basic 6.....	6
1.2.1	Variabler och tilldelning av objekt.....	6
1.3	Collections.....	7
1.3.1	Egenskaper	7
1.3.2	Metoder.....	7
1.3.3	Collections som bara accepterar en datatyp/klass.....	8
2	OBJEKTORIENTERAD PROGRAMMERING MED VISUAL BASIC	9
2.1	Klasser och objekt i Visual Basic 6.....	9
2.2	Klasser, moduler och formulär	9
2.2.1	Namn på filer i projekt	10
2.3	Egenskaper i klasser	10
2.4	Metoder i klasser.....	11
2.4.1	Accessmetoder	11
2.4.2	”Vanliga” metoder	12
2.4.3	Konstruktör och destruktör.....	14
2.5	Använda Class Builder för att skapa klass.....	14
2.5.1	Klassen bok	14
2.6	Arv av gränssnitt.....	16
2.6.1	Klassen IBil.....	17
2.6.2	Subklassen Taxi	18
2.6.3	Subklassen Lastbil.....	20
2.6.4	Test av klasserna IBil, Taxi och Lastbil.....	21
2.6.5	Kommentar till arv i Visual Basic.....	22
3	DATABASPROGRAMMERING MED ADO	23
3.1	ADO-modellen	23
3.1.1	Ett första exempel	24
3.2	Objektet Connection.....	25
3.2.1	Egenskapen ConnectString	25
3.2.2	Metoden Open.....	28
3.2.3	Metoden Execute.....	29
3.2.4	Metoden Close.....	30
3.3	Objektet Recordset.....	30
3.3.1	Egenskapen ActiveConnection	30
3.3.2	Egenskapen Source	31
3.3.3	Egenskapen CursorType.....	31
3.3.4	Egenskapen LockType.....	32
3.3.5	Egenskapen Fields.....	33
3.3.6	Metoden AddNew	34
3.3.7	Metoderna BOF och EOF.....	34
3.3.8	Metoden Close.....	34
3.3.9	Metoderna Move, MoveFirst, MovePrevious, MoveNext och MoveLast.....	35
3.3.10	Metoden Open.....	35
3.3.11	Metoden Update.....	35
3.4	Objektet Command (och Parameter)	36
3.4.1	Egenskapen ActiveConnection	36
3.4.2	Egenskapen CommandText	36
3.4.3	Egenskapen CommandType.....	36
3.4.4	Egenskapen Parameters.....	37
3.4.5	Metoden CreateParameter()	37
3.4.6	Metoden Execute().....	38
3.5	Skapa obundna Recordset-objekt	38
3.6	Konstruera SQL-satser i Visual Basic-kod.....	38
4	DATABASFUNKTIONER I KOD.....	41
4.1	Öppna tabell och hämta poster.....	41
4.1.1	Skapa Recordset-objekt och använda dess metod Open().....	41
4.1.2	Använda metoden .Execute() i Connection-objekt	41
4.1.3	(Skapa ett Connection-objekt implicit).....	42

4.2	Lägga till post i tabell	43
4.2.1	Använda Recordset-objekt	43
4.2.2	Använda SQL-sats och metoden Execute() i Connection-objekt	44
4.3	Uppdatera post i tabell	44
4.3.1	Använda Recordset-objekt	44
4.3.2	Använda SQL-sats och metoden Execute() i Connection-objekt	45
4.3.3	Använda Command-objekt och parametrar	45
5	FELHANTERING	47
5.1.1	Att inte hantera fel.....	47
5.1.2	Att prova om programsats utfördes korrekt.....	47
5.1.3	Använda sig av felhanterare i metod.....	48
5.1.4	Att hantera vissa fel men inte andra.....	49
5.2	Innan felhantering implementera	49
5.3	Lite om felsökning	49
6	ÖVRIGT.....	51
6.1	Tekniska detaljer	51
6.2	Komponenter och databaser.....	51
6.3	Lagrade frågor	51
6.4	Några vanliga fel.....	51
6.4.1	Tilldelning av variabler	51
6.4.2	Uppdatering av tabell	52
6.4.3	ODBC-källor.....	52
6.4.4	ADO-versioner.....	52
6.5	Fortsatt läsning.....	52

1 Deklarera variabler och grundläggande strukturer

I detta kapitel tittar vi på grundläggande datatyper och objekt i Visual Basic 6. Hanteringen av dem bägge är snarlik, med det finns en del skillnader som är viktiga att ta hänsyn till, bl.a. vid tilldelning till en variabel. Vidare tittar vi på objekt av typen Collection, de dynamiska vektorerna, som bl.a. används av databaskomponenterna i ADO.

1.1 Grundläggande datatyper i Visual Basic

I Visual Basic 6 finns ett antal grundläggande datatyper:

- heltal: Byte, Integer, Long
- decimaltal: Decimal, Single, Double
- boolesk (sant/falskt): Boolean
- strängar: String
- valuta och datum: Currency, Date
- ”fritext”: Variant (i en variabel av typen Variant kan lagras alla övriga typer).
- egen definierade data typer (*user defined types*, UDT)

Egen definierade typer är typer som vi programmerare själva skapar utifrån de övriga typerna.

1.1.1 Deklarera variabler

Deklaration av variabler i Visual Basic kallas för att dimensionera (*dimension*), därav det reserverade ordet Dim i början på deklareringsraden. Variabler kan deklaras på två sätt: genom att ange datatyp med *As Datatyp* eller med hjälp av suffix. Om datatyp utelämnas kommer variabeln bli av typen Variant (vilket inte är en effektiv datatyp!).

```
Dim strText1 As String 'Sträng deklarerad med As Datatyp
Dim strText2$          'Sträng deklarerad med suffixet $
Dim vntText3          'Variant då datatyp ej angivits
```

För att underlätta identifiering av datatypen för en variabel kan man sätta ett prefix först i namnet på variabeln, t.ex. strängen (String) strEfternamn och heltalet (Integer) intÅrtal. Denna standard kommer att användas i denna sammanfattning.

Även om svenska tecken kan användas i variabelnamn så rekommenderas det inte. Heltalsvariabeln ovan bör alltså heta intAartal eller intArtal.

1.1.2 Variabler och tilldelning av enkla värden

Variabler deklarerade av de grundläggande (enkla) datatyperna tilldelas ett värde med hjälp av likhetstecknet (=).

```
Dim strText As String 'Deklarera variabel
strText = "En sträng" 'Tilldela värde till variabel
```

Observera att likhetstecknet även används för att jämföra om två värden är lika. Detta kan i vissa fall leda till att tilldelning sker istället för jämförelse! Om en If-sats inte fungerar som tänkt bör man kontrollera om tilldelning sker istället för jämförelse.

I Visual Basic finns möjligheten att kunna använda variabler utan att först deklarerar dem. För att kompilator ska kunna hitta fel lättare så bör inte denna möjlighet användas. Därför bör man ändra standardinställningarna i Visual Basics IDE¹ genom att välja **Options...** från Tools-menyn och se till att kryssrutan **Require Variable Declaration** är förbockad. (Som ett rekommenderat komplement kan man kontrollera att alla moduler/filer innehåller `Option Explicit` längst upp).

```
Option Explicit 'Ange att variabler måste deklarerars innan de används

Dim strText As String
strText = "En sträng"
```

1.2 Objekt i Visual Basic 6

1.2.1 Variabler och tilldelning av objekt

I motsats till variabler av de enkla datatyperna (i avsnittet ovan) så krävs att man sätter, vad som i Visual Basic kallas, en referens till objekt (en typ av pekare). Detta gör man genom att använda Set-operatorn². Referenser sätts oftast i samband med att objekt skapas eller då en funktion returnerar ett objekt.

För att skapa objekt använder vi New-operatorn

```
Dim adoConn As ADODB.Connection 'Deklarera ...
Set adoConn = New ADODB.Connection '... skapa objekt och sätt referens
```

I exemplet ovan så kommer variabeln `adoConn` att referera till `Nothing` tills variabeln sätts till att referera till ett objekt med `Set` (explicit skapande av objekt). D.v.s. vi kan vänta med att sätta variabeln till att referera till ett objekt tills vi behöver variabeln första gången. I Visual Basic kan vi också använda `New`-operatorn i samband med att vi deklarerar en variabel. Första gången vi använder oss av variabeln så kommer ett objekt att skapas och variabel sätts att referera till det nyskapade objektet (implicit skapande av objekt).

```
Dim adoConn As New ADODB.Connection 'Deklarera - och sätt referens

If adoConn Is Nothing Then 'Kontrollera om objekt inte existerar (är Nothing)
    'Kommer alltid vara sant - objekt skapas vid jämförelse ovan...
End If
```

Vi kan, med denna andra form av deklARATION, av "misstag" skapa objektet bara genom att använda objektet i t.ex. en jämförelse. Slutsatsen av detta är att vi bör explicit skapa referenser till objekt genom att använda den första formen av variabeldeklARATION i exemplet ovan. (Detta trots att exempel i denna sammanfattning kan komma att använda andra formen för att spara plats.)

Ett exempel där Set-operatorn måste användas är när metoden **.Execute()** i Connection-objektet returnerar ett resultat (mer om detta i kapitel 3):

¹ IDE = Integrated Development Environment, d.v.s. integrerad programmeringsmiljö.

² Set-operatorn har tagits bort i Visual Basic.NET. ©

```
...  
strConn = "Provider=MSDASQL.1;Data Source=ODBCBokning"  
adoConn.Open strConn  
strQuery = "SELECT * FROM tblPersonal"  
  'Metoden .Execute() returnerar ett resultat i form av ett  
  'Recordset-objekt och därmed måste Set-operator användas.  
Set adoRS = adoConn.Execute(strQuery)  
...
```

1.3 Collections

Ett Collection-objekt är en dynamisk vektor som kan utökas eller minskas efter behov. Till skillnad mot vanliga vektorer (*arrays*) så kan ett Collection-objekt lagra värden (och objekt) av olika typer (och klasser). Vektorn kan med fördel användas med loopen `For Each` och är också vanligt förekommande i Visual Basics ramverkssklasser som t.ex. ADO (databaskomponenter).

En viktig skillnad mellan "vanliga" vektorer och Collection-vektorer är att Collection är en klass, d.v.s. vi måste skapa en instans av en Collection-vektor innan vi kan använda den.

```
Dim colList As Collection      'Deklarerar variabel  
Set colList = New Collection   'Skapa Collection-objektet
```

1.3.1 Egenskaper

Egenskapen `.Count` returnerar antalet positioner i vektorn.

1.3.2 Metoder

Det finns främst 3 metoder för att manipulera ett Collection-objekt: `.Add()`, `.Remove()` och `.Item()`. Metoden `.Add()` tar upp till fyra parametrar, men bara den första är obligatorisk. Den första parametern är själva värdet (eller objektet) som ska lagras och den andra parametern är en nyckel (sträng) som även kan användas för att indexera i vektorn. De andra två parametrarna används för att ange före respektive efter vilket värde (objekt) det nya värdet (objektet) ska placeras (se mer i VB:s hjälp).

```
'Deklarerar variabler  
Dim colList As Collection  
Dim varItem As Variant      'Måste vara Variant för att fungera i  
                             'For Each...Next-loopen.  
  
'Skapa Collection-objektet  
Set colList = New Collection  
'Lägg till värden i Collection-objektet  
colList.Add 1  
colList.Add 2, "Två"        'Ange nyckel till "Två" för värdet  
'Loopa över Collection-objektet...  
For Each varItem In colList  
  '...och lägg till värden i listruta  
  List1.AddItem varItem  
Next varItem
```

För att loopa över en Collection-vektor använder vi som sagt lämpligen `For Each`-loopen. Loopen kommer att ske (en iteration utföras) en gång för varje position i vektorn och i varje iteration så kommer värdet på "aktuell position" att placeras i en variabel (`varItem` i exempel ovan) som sen kan användas inuti loopen.

```
Dim colList As Collection
```

```
Dim varItem As Variant      'Måste vara Variant för att fungera i For Each-loopen

For Each varItem In colList 'Hämta värde på "aktuell position" i vektor
    'använd värde
Next varItem
```

Metoden **.Remove()** tar ett index som parameter. Antingen kan detta index vara en siffra som anger ordningsnumret för värdet som ska tas bort eller, om nyckel har angivits för värdena, så kan parametern vara nyckeln för värdet som ska tas bort. Nycklar är inte känsliga för skillnad mellan stora och små bokstäver.

```
'Kod fortsätter från exempel ovan
colList.Remove "tvÅ"      'Ta bort värde mha nyckel
colList.Remove 1         'Ta bort värde mha indextal

For Each varItem In colList
    List1.AddItem varItem
Next varItem
```

För att hämta ett värde i Collection-objektet används metoden **.Item()**. Till metoden skickas ett index, antingen ett tal (med början på 1) eller en nyckel (sträng). Metoden **.Item()** är standardmetoden vilket innebär att metodnamnet inte behöver skrivas ut för att användas (se andra loopen i exempel nedan).³

```
For i = 1 To colList.Count
    List1.AddItem colList.Item(i)
Next i
'Åtkomst av värde med den kortare varianten
For i = 1 To colList.Count
    List1.AddItem colList(i)
Next i
```

I exempel ovan så visas hur vi kan använda egenskapen **.Count** för att utföra en For-loop (istället för For Each-loop).

1.3.3 Collections som bara accepterar en datatyp/klass

Man kan skapa en klass utifrån klassen Collection som har samma egenskaper som Collection-objekt men som bara accepterar en datatyp/klass, d.v.s. som en vektor. Detta görs enklast med verktyget Class Builder (mer om detta verktyg senare).

³ I Visual Basic kan man ange standardmetod (eller standaregenskap) för en klass (eller modul). När man vill använda sig av standardmetodens så behöver man inte skriva ut namnet på metoden utan endast objektet. Ett typiskt exempel är textrutor där egenskapen Text är en standardegenskapen. Om man har en textruta med namnet Text1 så kan man komma åt texten i textrutan genom att antingen skriva Text1.Text eller bara Text1. (Se ”default member in class” i hjälpen för Visual Basic för mer information.)

2 Objektorienterad programmering med Visual Basic

Visual Basic 6 är inte ett fulländat objektorienterat språk, men det har många av de kriterier som krävs för objektorienterad programmering. Visual Basic stödjer bl.a. **inkapsling** och **polymorfism** fullt ut. Visual Basic klarar dock endast **arv av gränssnitt** (*interfaces*) och inte arv av klasser (implementation/kod). Men detta kan i viss mån komma runt genom **aggregering** (del av) och **delegering**.

2.1 Klasser och objekt i Visual Basic 6

De termer för klasser som används i Visual Basic stämmer inte alltid riktigt överens med, eller så finns det inga motsvarigheter till, termerna i objektorienterade modelleringstekniker (OO-tekniker). Det vi i OO-tekniker kallar för attribut kallar Microsoft för **egenskaper** (*properties*) i Visual Basic (mer om dessa nedan). Något som båda dock har gemensamt är **metoder** (*methods*) och i Visual Basic implementeras metoder som procedurer (`Sub`, *subprocedure*) eller som funktioner (`Function`).

Något som dock är unikt för Visual Basic är **händelser** (*events*). En händelse uppstår om t.ex. ett visst kriterie uppfylls, så som att en variabel i ett objekt uppnår ett visst värde. Man kan då aktivera/”avfyra” en händelse (*raise an event*) i objektet som användaren av objektet kan reagera på. T.ex. så skickar Windows händelsen `MouseClicked` när användaren klickar med en musknapp. Och om användaren klickade på en kommandoknapp så fångar Visual Basic upp detta meddelande och utför kommandoknappens `Click`-metod. (Händelser behandlas inte i denna sammanfattning.)

Klasser i Visual Basic skapas med hjälp av klassmoduler som lagras i CLS-filer.⁴ Ett sätt att snabbt skapa en klass är att använda insticksmodulen (*add-in*) **Class Builder** som finns (kan finnas) på *Add-Ins*-menyn i Visual Basics IDE (se stycke om Class Builder nedan).

2.2 Klasser, moduler och formulär

Den största skillnaden mellan klasser, moduler och formulär i Visual Basic är att man kan skapa instanser (objekt) av klasser men inte moduler och formulär. Moduler och formulär innehåller bara kod som kan anropas från andra formulär, moduler och klasser – kort sagt: de är ett förråd av metoder och egenskaper. Formulär innehåller dock även kod som beskriver formuläret och dess kontroller. Trots denna skillnad mellan moduler och formulär så kommer termen modul användas för både moduler och formulär i denna sammanfattning.

Klasser är mallar som objekt skapas utifrån – endast metodernas kod är gemensamt för objekt av klassen. Värden i ett objekts (instans-)variabler (egenskaper) är inte detsamma som i ett annat objekts (instans-)variabler. Det finns endast ett värde för variabler i ett formulär eller en modul (eftersom det bara kan finnas en instans av ett formulär eller modul.)

Det finns dock saker som är gemensamma för klasser och moduler. Båda kan deklarerat metoder och egenskaper som publika (`Public`) eller privata (`Private`). Likaså är båda självständiga filer som kan användas i flera Visual Basic-projekt. I Visual Basic kan man numera även ha metoder som endast är åtkomliga från klasser i samma Visual Basic-projekt, s.k. Friend-metoder. Dessa deklarerar med det reserverade ordet `Friend` istället för `Public` eller `Private`.

⁴ Några andra filer i Visual Basic är projektfiler (.VBP), formulär (.FRM) och moduler (.BAS).

```
Public Sub PublikProcedur()      'Denna procedur kan anropas av ...
    'Utför något                '... alla andra metoder
End Sub

Private Function PrivatFunktion() 'Denna funktion kan endast anropas av ...
    'Utför något                '... andra metoder i modulen
End Function

Friend Sub IModulenPublikProcedur() 'Denna procedur kan endast anropas av ...
    'Utför något                '... andra metoder i projektet
End Sub
```

2.2.1 Namn på filer i projekt

Även om filerna i Visual Basic-projekt visar vilken typ av fil det är så kan det vara lämpligt att ge filnamnen ett prefix, bl.a. för att enkelt kunna skilja mellan t.ex. formulär och klasser med samma namn. Det blir även lättare att hitta filerna i Utforskaren – t.ex. kan man ge klasser, moduler och formulär prefixen `cls`, `mod` respektive `frm`. Filnamn för klassen `Person` och formuläret, som används för att redigera objekt av klassen, blir då `clsPerson.cls` respektive `frmPerson.frm`.

2.3 Egenskaper i klasser

En egenskap motsvarar, som nämnts ovan, av attribut i OO-tekniker. En egenskap kan implementeras på ett av två sätt: som en publik variabel som nås direkt eller som en privat variabel som endast kan nås via metoder, s.k. accessmetoder. Den publika variabeln har samma namn som egenskapen. Genom att använda en publik variabel så har användaren av klassen full tillgång till variabeln, d.v.s. programmeraren kan både läsa och skriva till variabeln. Detta sätt är inte att rekommendera!

Nedan deklarerar egenskapen `Namn` i klassen `Person` som en publik variabel.

```
Public Namn As String          'Publik variabel för egenskapen Namn i klassen Person
... 'Resten av klassen Person
```

För att sätta och hämta värdet på egenskapen `Namn` i ett objekt av klassen `Person` används följande kod:

```
Dim objPerson As New Person
objPerson.Namn = "Björn Persson"
MsgBox "Personens namn är: " + objPerson.Namn
```

Om man istället använder accessmetoder för att implementera egenskaper så kan man lägga in tester i accessmetoden för att t.ex. förhindra ogiltiga värden på egenskapen. Man kan också göra variabeln endast läsbara genom att inte implementera en metod för att skriva till den privata variabeln. Den privata variabeln brukar namnges genom att sätta ett prefix före egenskapens namn och accessmetoderna får då samma namn som egenskapen.⁵ T.ex. får egenskapen `Namn` variabelnamnet `mstrNamn` och egenskapen `Fodelsear` får variabelnamnet `mintFodelsear`. (Se mer stycket *Accessmetoder* nedan.)

⁵ Om den privata variabeln skulle ha samma namn som egenskapen så skulle inte Get-metoden för att hämta värdet fungera!

2.4 Metoder i klasser

Bortsett från att metoder kan implementeras som publika och privata procedurer eller funktioner så finns det två typer av metoder i Visual Basic-klasser: accessmetoder och ”vanliga” metoder

2.4.1 Accessmetoder

En del av objektorienterad programmering är inkapsling, d.v.s. att egenskaper (variabler) i objekt endast ska kunna nås via metoder. För att stödja detta i Visual Basic använder man sig av accessmetoderna **Get()**, **Let()** och **Set()**.

Metoden **Get()** används, som det låter på namnet, för att hämta/returnera både enkla datatyper och objekt. Däremot så skiljer Visual Basic på tilldelning av enkla datatyper och objekt.⁶

Metoden **Let()** används för att tilldela enkla datatyper medan **Set()** används för att tilldela objekt. Nedan följer exempel med accessmetoderna i klassen `Person` där egenskapen `Namn` är en sträng (d.v.s. enkel datatyp) och `Bil` är ett objekt av klassen `Bil`.

```
Option Explicit
'Deklarera medlemsvariabler (egenskaper/attribut)
Dim mstrNamn As String 'Medlemsvariabler brukar föregås av bokstäverna mvar
Dim mobjBil As Bil    ' för att tala om att det är en medlemsvariabler
                    ' (Class Builder lägger t.ex. till dem).
                    ' Har redigerat för att ange datatyp för variabler.

'Accessmetoder för Namn
Property Let Namn(text As String)
    mstrNamn = text
End Property

Property Get Namn() As String
    Namn = mstrNamn
End Property

'Accessmetoder för Bil
Property Set Bil(obj As Bil)
    Set mobjBil = obj 'Set då objekt
End Property

Property Get Bil() As Bil
    Set Bil = mobjBil 'Set då objekt
End Property
```

För att tilldela eller läsa (hämta) en egenskap i ett objekt så används punktnotation, d.v.s.

```
Objekt.Egenskap = "värde" 'Tilldela värde till egenskap
enVariabel = Objekt.Egenskap 'Läsa/hämta värde från egenskap
```

Notationen för att använda accessmetoderna lurar ofta VB-programmerare att tro att de tilldelar värdena direkt till publika instansvariabler i objektet. I nedanstående kod anropas accessmetoderna i klassen ovan (d.v.s. i ett objekt av klassen `Person`) genom ”vanlig” tilldelning med likhetstecknet (“=”).

```
'Deklarera variabler
Dim objPerson As New Person
```

⁶ Denna egenhet har försvunnit i Visual Basic.NET. ©

```
Dim objBil As New Bil
Dim strMeddelande As String
  'Använd Let- och Set-metoder i Person-objekt
objPerson.Namn = "Sture"           'Tilldelning av vanlig datatyp med Let-metod
Set objPerson.Bil = objBil        'Tilldelning av objekt med Set-metod
  'Använd Let-metod i Bil-objekt
objBil.Marke = "Saab"
  'Använd Get-metoder i både Person- och Bil-objekt för att
  'bygga en sträng som skickas till en meddelanderuta
strMeddelande = "Personen " + objPerson.Namn
strMeddelande = strMeddelande + " äger en bil av märket "
strMeddelande = strMeddelande + objPerson.Bil.Marke
  'Visa strängen mha meddelanderuta
MsgBox strMeddelande
```

I den nästsista raden kod anropas först Person-objektets Get-metod för egenskapen Bil (objPerson.Bil) vilket resulterar i att ett objekt av klassen Bil returneras. Sedan anropas Get-metoden i Bil-objektet (det sista .Marke). Koden kunde ha skrivits om enligt följande (endast relevant kod visas – övrigt ersatt med tre punkter):

```
  'Deklarera variabler
...   'Kod bortklippt!!
Dim strMeddelande As String
Dim objBil2 as Bil           'Ny variabel behövs för mellansteget nedan
...   'Kod bortklippt!!
strMeddelande = strMeddelande + " äger en bil av märket "
Set objBil2 = objPerson.Bil  'Använd Get-metod i Person-objekt
strMeddelande = strMeddelande + objBil2.Marke  'Anv Get i Bil-obj
  'Visa strängen mha meddelanderuta
MsgBox strMeddelande
```

2.4.2 "Vanliga" metoder

Precis som med accessmetoder så anropas metoder med punktnotation. Vi lägger till en metod **Spara()** till klassen Person för att spara objektet till fil. I metoden kommer vi att öppna en ny fil (d.v.s. skriva över en eventuell gammal fil) och skriva namnet på personen till filen. Lägg till nedanstående kod efter sista accessmetoden.

```
Public Sub Spara(Filnamn As String)
  Open Filnamn For Output As #1 'Öppna fil för att skriva till (ny fil skapas!)
  Print #1, mvarNamn           'Skriv eget namn till filen
  Close #1                     'Stäng filen

  'Spara endast bil om person äger en bil, d.v.s. bil existerar
  If Not mvarBils Is Nothing Then
    mvarBil.Spara Filnamn      'Be Bil-objekt att spara sig själv
  End If
End Sub
```

Eftersom bilen, som personen äger, är ett objekt så måste vi be Bil-objektet själv att spara sig själv i metoden ovan. Därför lägger vi till en metod **Spara()** även i klassen Bil. Metoden i klassen Bil kommer dock endast att anropas av objekt av klassen Person och kan därför vara en Friend-metod. Metoden öppnar en fil för att lägga till på slutet (så att vi inte skriver över personens namn) och skriver sitt märke till filen.

```
Friend Sub Spara(Filnamn As String)
  Open Filnamn For Append As #1  'Öppna fil för skriva (lägger till sist!)
  Print #1, mvarMarke           'Skriv märke till filen
  Close #1                      'Stäng filen
End Sub
```

Koden för att anropa metoden **Spara()** i objekt av klassen Person (som i sin tur anropar **Spara()** i objektet av klassen Bil) blir därmed följande:

```
objPerson.Spara "C:\Student\PersBil.txt" 'Spara till filen PersBil.txt
```

Resultatet i filen PersBil.txt blir:

```
Sture  
Saab
```

Som egen övning lämnas metoderna **LasIn()** (läs in) i båda klasser.

2.4.2.1 Parametrar till metoder

Parametrar till metoder kan skicka som värden (*by value*) eller via referens (*by reference*). Om ett argument skickas som ett värde så kommer en kopia på värdet att skickas. Detta innebär att om jag skickar en variabel som ett argument till en metod så kommer en kopia av värdet i variabel att skickas. D.v.s. variabeln är opåverkad av eventuella ändringar av värdet i metoden. Som argument till metoden kan både literaler (konstant värde) eller variabler skickas.

Men om en parameter i en metod skickas via en referens så kan värdet i en variabel ändras. Argumentet till metoden måste vara en variabel (som i kommande exempel), d.v.s. literaler får inte skickas.

```
'Metod som tar emot två parametrar som värden - argument kan vara literaler  
'eller variabler  
Private Sub BytVarde(ByVal a As Integer, ByVal b As Integer)  
    Dim temp As Integer  
    temp = a  
    a = b  
    b = temp  
End Sub  
  
'Metod som tar emot två parametrar som referenser - argument måste vara  
'variabler  
Private Sub BytReferens(ByRef a As Integer, ByRef b As Integer)  
    Dim temp As Integer  
    temp = a  
    a = b  
    b = temp  
End Sub
```

I nedanstående exempel testas metoderna ovan. Observera att värdena i variablerna `tempa` och `tempb` ändras endast då metoden `BytReferens` anropas.

```
Private Sub Command1_Click()  
    Dim tempa As Integer, tempb As Integer  
  
    tempa = CInt(Text1.Text)  
    tempb = CInt(Text2.Text)  
    BytVarde tempa, tempb  
    Text3.Text = tempa  
    Text4.Text = tempb
```

```
tempa = CInt(Text1.Text)
tempb = CInt(Text2.Text)
BytReferens tempa, tempb
Text5.Text = tempa
Text6.Text = tempb
End Sub
```

2.4.2.2 Distribuerade komponenter och parametrar till metoder

När vi har parametrar till metoder bör vi endast använda `ByRef` om argument till metod kommer ändras eller om vi förväntar oss ett värde i retur från metod. I alla andra fall bör vi använda `ByVal` för att inte slösa på onödiga resurser på att skicka samma värde tillbaka.

2.4.3 Konstruktör och destruktör

Även klasser i Visual Basic har en konstruktör och destruktör – `Class_Initialize()` respektive `Class_Terminate()`. `Class_Initialize()` exekveras då objektet skapas och `Class_Terminate()` när alla referenser till objekt har försvunnit. En viktig skillnad mot många andra objektorienterade språk är att konstruktörer i Visual Basic inte kan ta emot några parametrar (och därmed inte heller överbelastas).⁷

2.5 Använda Class Builder för att skapa klass

Class Builder är ett verktyg med grafiskt gränssnitt för att skapa grundstommen till klasser i Visual Basic. Verktøget används främst för att skapa nya klasser eller uppdatera klasser skapade av Class Builder. Dock kan, med risk för fel, verktyget även användas för att uppdatera klasser skapade för hand. Class Builder nås från Add-Ins-menyn med alternativet **Class Builder Utility...**

2.5.1 Klassen bok

Som exempel på hur Class Builder fungerar ska vi skapa en klass `Bok` med egenskaperna `Titel`, `Författare` och `Förlag`. Även om Visual Basic tillåter variabler med svenska tecken (å, ä och ö) så kommer vi att ta bort prickarna och cirkelarna över svenska tecken för att vara på den säkra sidan. D.v.s. egenskaperna kommer att få namnen `Titel`, `Forfattare` och `Forlag`. Vi lägger även till två metoder `LånaUt()` och `LämnaTillbaka()` (d.v.s. `LanaUt()` resp. `LamnaTillbaka()`).

1. Starta Visual Basic och skapa ett nytt projekt, t.ex. en standard EXE.
2. Från Project-menyn, välj **Project1 Properties...** och ändra projektets namn i textrutan **Project Name** (till t.ex. `ClassBuilderTest`). Klicka på OK för att stänga dialogrutan.
3. Starta Class Builder från Add-Ins-menyn. Finns inte alternativet **Class Builder Utility...** på menyn måste du lägga till den med **Add-In Manager...** Markera alternativet **VB 6 Class Builder Utility** och kryssa i kryssrutan **Loaded/Unloaded** nere till höger i dialogrutan.⁸ Stäng dialogrutan Add-In Manager och starta Class Builder från Add-Ins-menyn.
4. Markera projektets namn (`ClassBuilderTest` om du ändrade projektets namn till det i punkt 2 ovan).

⁷ I Visual Basic.NET så kan nu konstruktörer ta emot parametrar.

⁸ Skulle inte VB 6 Class Builder Utility finnas som alternativ i dialogrutan Add-In Manager behöver du antagligen installera om Visual Basic/Studio. Du kan, i installationsprogrammet för Visual Basic/Studio, välja att installera om alla filer.

5. Klicka på den mest vänstra ikonen (**Add New Class**) eller välj **New->Class...** från File-menyn. Fyll i "Bok" i textrutan **Name** och klicka på OK.
6. Markera klassen Bok och klicka på den tredje ikonen från vänster (**Add New Property to Current Class**) eller välj **New->Property...** från File-menyn för att lägga till en ny egenskap (nytt attribut) till klassen. Fyll i "Titel" i textrutan **Name** och välj "String" som datatyp i komborutan **Data Type**. Kontrollera att radioknappen **Public Property (Let, Get, Set)** är markerad. Klicka OK för att spara egenskapen. Upprepa denna punkt för egenskaperna **Forfattare** och **Forlag** – välj datatypen "String" för bägge.
7. Markera klassen Bok och klicka på den fjärde ikonen från vänster (**Add New Method to Current Class**) eller välj **New->Method...** från File-menyn för att lägga till en ny metod i klassen. Fyll i "LanaUt" (låna ut) i textrutan **Name**. Klicka på plustecknet till höger i dialogrutan för att lägga till en parameter (*argument*) till metoden. Fyll i "Datum" i textrutan **Name** för att namnge parametern och se till att datatypen är "Variant" i komborutan **Data Type** (förklaring varför inte datatypen Date kommer senare). Klicka på OK för att stänga dialogrutan **Add Argument** och sen OK en gång till för att stänga dialogrutan **Method Builder**.
8. Lägg till en metod till med namnet "LamnaTillbaka" (lämna tillbaka) utan parametrar.
9. Välj **Update Project** från File-menyn och stäng sen Class Builder.

Det ska nu finnas en klass Bok med följande (eller liknande ☺) kod i:

```
Option Explicit

'local variable(s) to hold property value(s)
Private mvarTitel As String 'local copy
Private mvarForfattare As String 'local copy
Private mvarForlag As String 'local copy

Public Sub LamnaTillbaka()
End Sub

Public Sub LanaUt(Datum As Variant)
End Sub

Public Property Let Forlag(ByVal vData As String)
'used when assigning a value to the property, on the left side of an assignment.
'Syntax: X.Forlag = 5
    mvarForlag = vData
End Property

Public Property Get Forlag() As String
'used when retrieving value of a property, on the right side of an assignment.
'Syntax: Debug.Print X.Forlag
    Forlag = mvarForlag
End Property

Public Property Let Forfattare(ByVal vData As String)
'used...
    mvarForfattare = vData
End Property

Public Property Get Forfattare() As String
'used...
    Forfattare = mvarForfattare
End Property

Public Property Let Titel(ByVal vData As String)
'used...
    mvarTitel = vData
```

```
End Property

Public Property Get Titel() As String
    'used...
    Titel = mvarTitel
End Property
```

Som syns i koden så har Class Builder lagt till kod i accessmetoderna så att egenskapernas värden kan sättas och läsas. Nu återstår bara att fylla i kod i de två metoderna `LanaUt()` och `LammaTillbaka()`. För att de ska fungera måste vi lägga till en (privat) instansvariabel `DatumUtlånad` (datum utlånad) av datatypen `Variant`. Variabeln deklarerar vi längst upp i tillsammans med de andra tre variablerna:

```
Private mvarDatumUtlånad As Variant
```

Metoderna ser ut enligt följande:

```
Public Sub LanaUt(Datum As Variant)
    mvarDatumUtlånad = Datum
End Sub

Public Sub LammaTillbaka()
    mvarDatumUtlånad = Null
End Sub
```

Orsaken till att vi lagrar datumet i en variabel av typen `Variant` är för att ett datum aldrig kan vara noll eller inget (`Null`).⁹ Däremot kan en variabel av typen `Variant` lagra ett datum och sättas till noll (eller `Null`). Och i exempelklassen ovan så kan vi sätta variabeln `DatumUtlånad` till `Null` när boken lämnas tillbaka. Om någon sen vill låna boken så kan vi kontrollera om värdet är `Null` för att se om boken inte är utlånad (och därmed tillgänglig). Vi skulle alltså kunna lägga till metoden `Utlånad` (utlånad) till klassen `Bok` med följande kod:

```
Public Function Utlånad() As Boolean
    'Svarar med sant om boken är utlånad
    If mvarDatumUtlånad = Null Then
        Utlånad = False
    Else
        Utlånad = True
    End If
End Function
```

2.6 Arv av gränssnitt

Som nämnt tidigare så kan klasser endast ärvta gränssnitt (en annan klass publika egenskaper) i Visual Basic. Det är inte riktigt rakt på sak att ärvta och implementera ett gränssnitt, så därför visas det genom ett exempel. Vad vi vill kunna göra är följande:

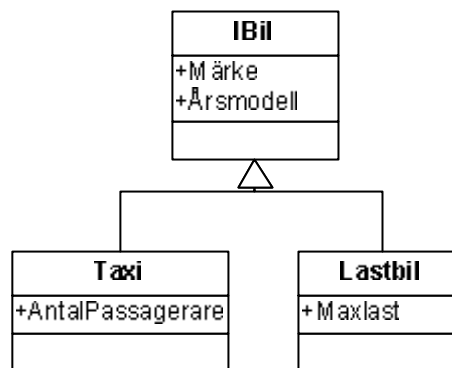
```
'Deklarera variabler
Dim objBil As IBil      'Deklarera variabel av superklassen (utan att skapa)
Dim objTaxi As New Taxi 'Deklarera och skapa variabel av subklassen
```

⁹ Jag testade att skriva ut en oinitierad variabel av typen `Date` och fick som resultat 99-12-30 under Windows 2000 (koden var `Dim d As Date ↵ MsgBox Format(d, "yy-mm-dd")`).


```
objTaxi.Marke = "Saab" 'Ange märke för taxi
Set objBil = objTaxi 'Säg till bilen att referera till taxin
'Fråga bilen vad den har för märke - visa i meddelanderuta
MsgBox "Bilens märke är: " & objBil.Marke
```

Detta är ett exempel på polymorfism, d.v.s. att vi kan referera till en instans av subklass m.h.a. en variabel av superklassens typ. I exemplet nedan kommer vi att skapa lite testkod som ger möjlighet till att skapa objekt av två subklasserna och sen skriva ut bägge med samma kod. I exemplet ges superklassen (utan implementation) namnet `IBil` för att visa att det är ett gränssnitt (enligt Microsofts namngivningsstandard).

Alla bilar har egenskaperna `märke` och `årsmodell`, vilket kan samlas i klassen `Bil`. Och klasserna `Taxi` och `Lastbil` kan genom arv utöka klassen `Bil` med egenskaperna `antalPassagerare` respektive `maxlast`. Vi får alltså följande modell (se bild till höger):



För att skapa klasser skapar vi ett nytt projekt i Visual Basic – välj **Standard EXE**.

1. Namnge projektet genom att välja **Project1 Properties...** från Project-menyn och fylla i namnet på projektet, t.ex. `ArvBil`, i textrutan **Project Name**.
2. Skapa sen de tre klasserna genom att välja **Add Class Module** från Project-menyn (tre gånger) och sen klicka OK i dialogrutan som visas. Döp om klasserna till `IBil`, `Taxi` respektive `Lastbil` genom att ändra egenskapen (**Name**) i Properties-fönstret för respektive klassmodul.
Vill du så kan du använda Class Builder för att skapa klasserna, men då bör en del kod (som inte används) raderas och annan kod läggas till.
3. Fyll sen i koden i styckena nedan.

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

2.6.1 Klassen `IBil`

Klassen `IBil` specificerar endast vilka metoder som objekt av klassen `IBil` och dess subklasser ska svara på – därför är metoderna tomma. Denna klass är annars inga konstigheter med. Vi lämnar metoderna tomma då vi ändå inte kan ärva koden.

```
Option Explicit
'Variabler kan saknas då accessmetoderna visar på vilka egenskaper som klassen
' har.

'Metoder kan vara tomma eftersom den ska implementeras i subklasser
Public Property Let Marke(ByVal vData As String)
    '
End Property

Public Property Get Marke() As String
    '
End Property

Public Property Let Arsmodell(ByVal vData As Integer)
```

```
End Property  
  
Public Property Get Arsmodell() As Integer  
  
End Property
```

(I metoderna har en kommentar, d.v.s. ett enkelt citattecken, placerats för att Visual Basic inte ska ta bort metoderna av misstag. Detta sker ibland med metoder som inte innehåller någon kod.)

2.6.2 Subklassen Taxi

Som nämnt ovan så är det inte riktigt rakt på sak att ärva ett gränssnitt från en superklass.

- Steg 1 är att vi måste ange att vi ärver gränssnittet från en annan klass.
- Steg 2 är att deklarerar instansvariabler för alla egenskaper – subklassens egna och de egenskaper den ärver från en superklass.
- I steg 3 implementera vi metoder med samma namn som i gränssnittet vi ärver ifrån. Om vi inte gör det så kommer inte en variabel av subklassen kunna anropa metoderna – vi måste då ha en variabel av superklassen för att kunna anropa metoderna.¹⁰
- Sista steget, steg 4, är att implementera superklassens metoder – vilket inte är helt självklart.

I ”rent” objektorienterade språk, som C++ och Java, hade det räckt med steg 1 och delvis steg 2 – vi ärver både gränssnittet, instansvariabler och metodernas implementation. Om vi vill definierat om en eller flera metoder hade vi dock skapat en metod i subklassen med samma namn som i superklassen, vilket innebär att även steg 3 behövs.

Steg 1: Vi anger att klassen `Taxi` ärver från klassen `IBil` med det reserverade ordet `Implements`, vilket skrivs längst upp i klassen (efter `Option Explicit`), med superklassen efter (`IBil` i vårt fall).

```
Option Explicit  
Implements IBil 'Ange att vi vill ärva gränssnittet från klassen Bil
```

Steg 2: Vi deklarerar variabler och göra Let-/Get-metoder för egenskaperna (i vårt exempel egenskaperna `Marke`, `Arsmodell` och `AntalPassagerare`).¹¹

```
Private mvarMarke As String 'Variabel för att hålla reda på IBils egenskap  
Private mvarArsmodell As Integer 'Variabel för att hålla reda på IBils egenskap  
Private mvarAntalPassagerare As Integer 'Var. för att hålla reda på egen egenskap
```

Steg 3: Implementera subklassens egna metoder samt ärvda metoder från superklassen. Här implementerar vi alltså accessmetoder för subklassens egna egenskaper men även egenskaperna som vi ärvt från superklassen (vilket är enligt princip visad i tidigare exempel).

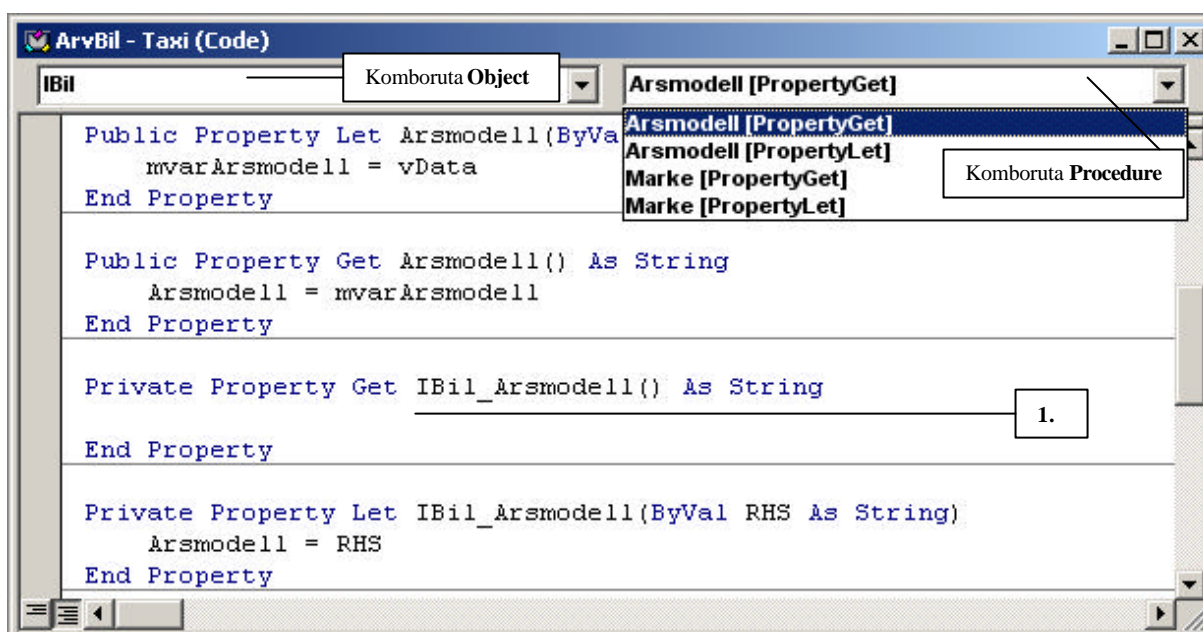
¹⁰ Detta är lite förvirrande, men det kommer sig av hur Visual Basic fungerar. När vi ärver ett gränssnitt i Visual Basic så ger vi objekt av subklassen ett gränssnitt till utöver sitt eget, d.v.s. superklassens gränssnitt. Och metoderna i superklassens gränssnitt måste anropas via en variabel av superklassens typ.

¹¹ Let-/Get-metoderna är utelämnade här men finns i den fullständiga koden som följer.

Vi måste alltså implementera accessmetoder för egenskaperna Marke, Arsmodell och AntalPassagerare. (Se nedan för kod.)

Steg 4: Och sist måste vi implementera superklassens metoder så att polymorfism fungerar. Det är detta som gör att arv och polymorfism inte är rakt på sak i VB (i t.ex. Java så är detta steg överflödigt). För att implementera superklassens (IBil) metoder så gör man följande:

1. Öppna kodfönstret för subklassen.
2. Välj superklassen (IBil) i komborutan **Object** till vänster (se bild).



3. Välj sen metod att implementera i komborutan **Procedure** till höger (se bild ovan) för att skapa ”skalkod” för metoden (se punkt 1. i bild ovan).
4. Fyll i kod för att anropa den egna metoden för denna ”ärvda” metod. I exemplet i bilden ovan anropar vi klassen Taxis Get-metod för egenskapen Arsmodell, d.v.s. koden blir `IBil_Arsmodell = Arsmodell`.

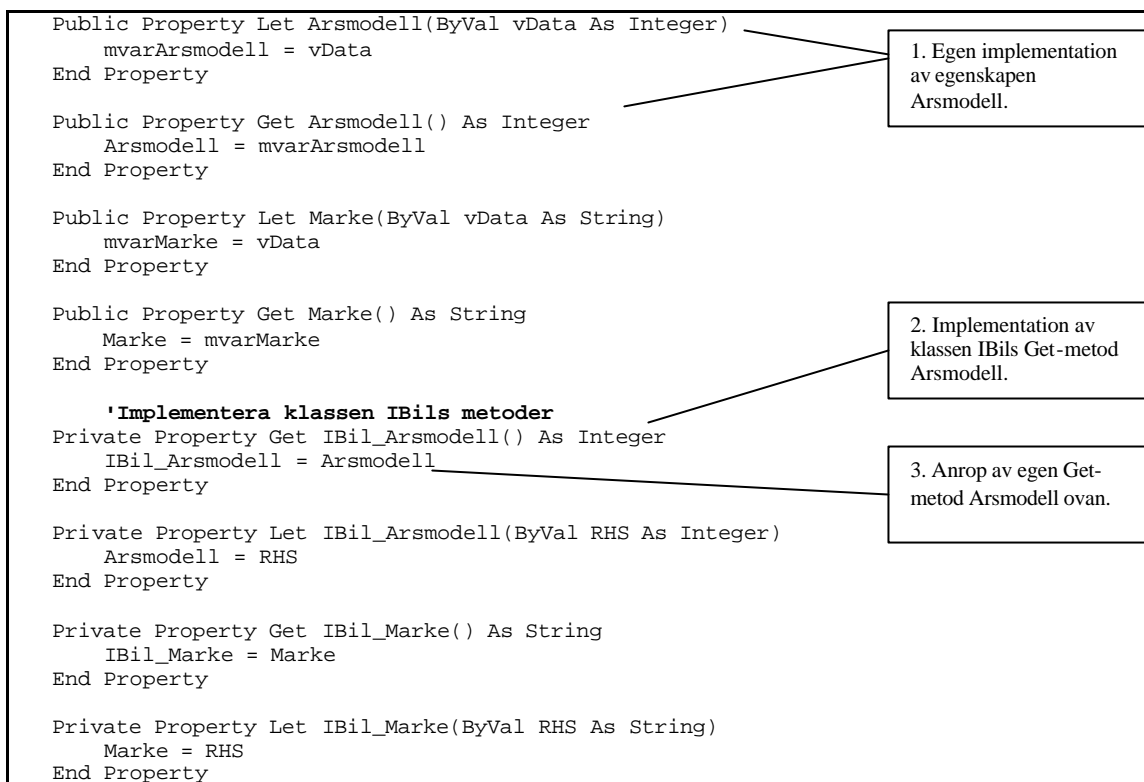
Som sagt, det är inte riktigt rakt på sak att implementera arv av gränssnitt i Visual Basic. Nedan följer den kompletta koden för klassen Taxi:

```
Option Explicit
Implements IBil 'Ange att gränssnitt ska ärvas från Bil

'Deklarera variabler för egenskaperna
Private mvarAntalPassagerare As Integer
Private mvarArsmodell As Integer
Private mvarMarke As String

'Accessmetoder för egenskaper i klassen
Public Property Let AntalPassagerare(ByVal vData As Byte)
    mvarAntalPassagerare = vData
End Property

Public Property Get AntalPassagerare() As Byte
    AntalPassagerare = mvarAntalPassagerare
End Property
```



(De konstiga namnen på parametrarna [RHS] har skapats av Visual Basic själv då man anger att man ska implementera en metod från det ärvda gränssnittet.)

2.6.3 Subklassen Lastbil

Klassen Lastbil ser likadan ut som klassen Taxi, med skillnaden att variabeln `mvarAntalPassagerare` (samt dess Let-/Get-metoder) byts ut mot `mvarMaxlast` och motsvarande Let-/Get-metoder. Klassen Lastbils kod som skiljer sig från klassen Taxis visas nedan:

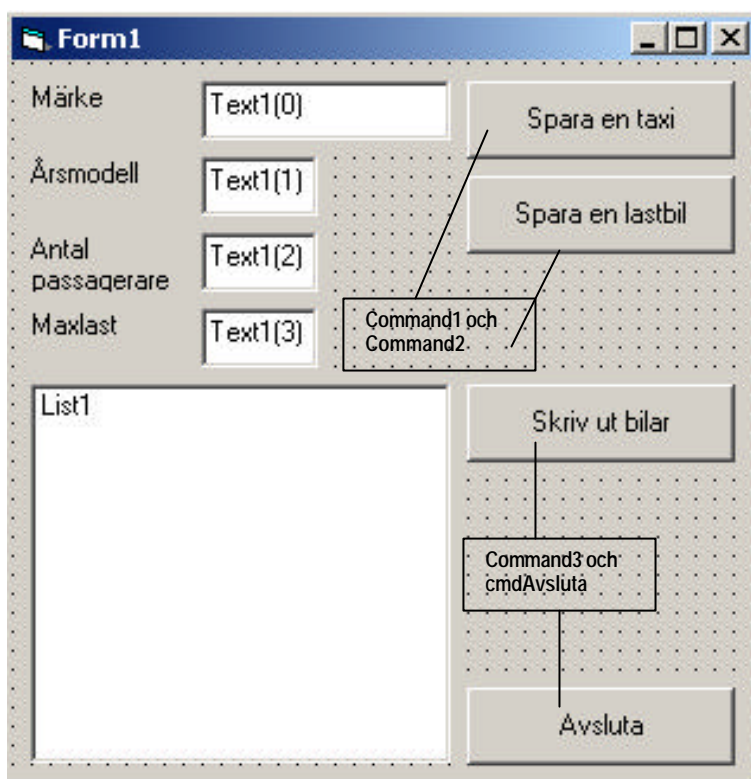
```
...
    'Deklarera variabler för attributen
Private mvarMaxlast As Integer
...
Public Property Let Maxlast(ByVal vData As Integer)
    mvarMaxlast = vData
End Property

Public Property Get Maxlast() As Integer
    Maxlast = mvarMaxlast
End Property
...
```

2.6.4 Test av klasserna IBil, Taxi och Lastbil

För att testa klasserna IBil, Taxi och Lastbil placerar vi följande kontroller på formuläret Form1:

- 4 st. etiketter (se bild)
- 4 st. textrutor i en *control array* (placera en textruta, med standardnamnet Text1, på formuläret, kopiera textrutan och klistra sen in textrutan tre gånger. När textrutan klistras in första gången kommer VB fråga om du vill göra en *control array*, vilket du svara Ja på.)
- 1 st. listruta med namnet List1.
- 4 st. kommandoknappar med namnen Command1, Command2, Command3 resp. cmdAvsluta.



Fyll sedan i koden nedan i formulärets kodfönster:

```
Option Explicit
'Deklarera variabler
Dim colBilar As Collection
Dim objTaxi As Taxi
Dim objLastbil As Lastbil
Dim objBil As IBil

' Detta måste vara en global variabler, men ...
' ... övriga variabler har deklarerats globalt här
' för att spara plats (och därmed papper). Detta är
' INTE en bra idé - variabler bör deklarerars i
' respektive metod där de används!

Private Sub cmdAvsluta_Click()
    'Knapp för att avsluta
    'Avsluta programmet genom att ladda ur formuläret
    Unload Me
    'Snyggare avslut än att bara skriva End, vilket också fungerar
End Sub

Private Sub Command1_Click()
    'Knapp för att spara en taxi
    Set objTaxi = New Taxi
    'Skapa objekt av klassen Taxi

    objTaxi.Märke = Text1(0)
    objTaxi.Arsmodell = Text1(1)
    objTaxi.AntalPassagerare = Text1(2)
    colBilar.Add objTaxi
    Set objTaxi = Nothing
    'Sätt värden i objekt
    'Lägg i Collection-vektorn
    'Sätt variabeln att peka på inget
End Sub

Private Sub Command2_Click()
    'Knapp för att spara en lastbil
    Set objLastbil = New Lastbil
    'Skapa objekt av klassen Lastbil

    objLastbil.Märke = Text1(0)
    objLastbil.Arsmodell = Text1(1)
    objLastbil.Maxlast = Text1(3)
    colBilar.Add objLastbil
    Set objLastbil = Nothing
    'Sätt värden i objekt
    'Lägg i Collection-vektorn
    'Sätt variabeln att peka på inget
End Sub
```

```
Private Sub Command3_Click()                'Knapp för att skriva ut bilar
    'Loopa över vektorn och skriv ut bilar i listrutan
    For Each objBil In colBilar             'objBil är variabel av superklassen
        List1.AddItem objBil.Marke + ", " + Str(objBil.Arsmode)
    Next
End Sub

Private Sub Form_Load()                    'Metod som anropas då formulär laddas
    Dim i As Integer
    'Töm textrutor på text
    For i = 0 To 3
        Text1(i) = ""
    Next i
    'Skapa vektorn att lagra bilar i
    Set colBilar = New Collection
End Sub
```

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

Vill ni veta av vilken typ dom olika bilarna är samt deras egna egenskaper så kan ni ändra i formulärets metod `Command3_Click()` så att koden blir följande:

```
Private Sub Command3_Click()                'Knapp för att skriva ut bilar
    Dim strTyp As String
    Dim strEgen As String
    'Loopa över vektorn och skriv ut bilar i listrutan
    For Each objBil In colBilar
        If TypeOf objBil Is Taxi Then      'Testa om det är ett Taxi-objekt
            Set objTaxi = objBil
            strTyp = "Taxi: "
            strEgen = "Antal passagerare: " + Str(objTaxi.AntalPassagerare)
        ElseIf TypeOf objBil Is Lastbil Then 'Testa om det är ett Lastbil-objekt
            Set objLastbil = objBil
            strTyp = "Lastbil: "
            strEgen = "Maxlast: " + Str(objLastbil.Maxlast)
        End If
        'Lägg till bilarna i listrutan
        List1.AddItem strTyp + objBil.Marke + ", " + _
            + Str(objBil.Arsmode) + ", " + strEgen
    Next
End Sub
```

- ① Observera att nästan alla variabler har deklarerats globalt (i formulär) i exempelkod i detta avsnitt. Detta har gjorts för att spara plats (och därmed papper). Detta är alltså **inte** att rekommendera – variabler bör endast deklarerars i den räckvidd där de används, d.v.s. med så liten räckvidd som möjligt. Den enda variabeln som måste vara global ovan är vektorn som används för att lagra instanser av klasserna och då den används i de flesta av metoderna.

2.6.5 Kommentar till arv i Visual Basic

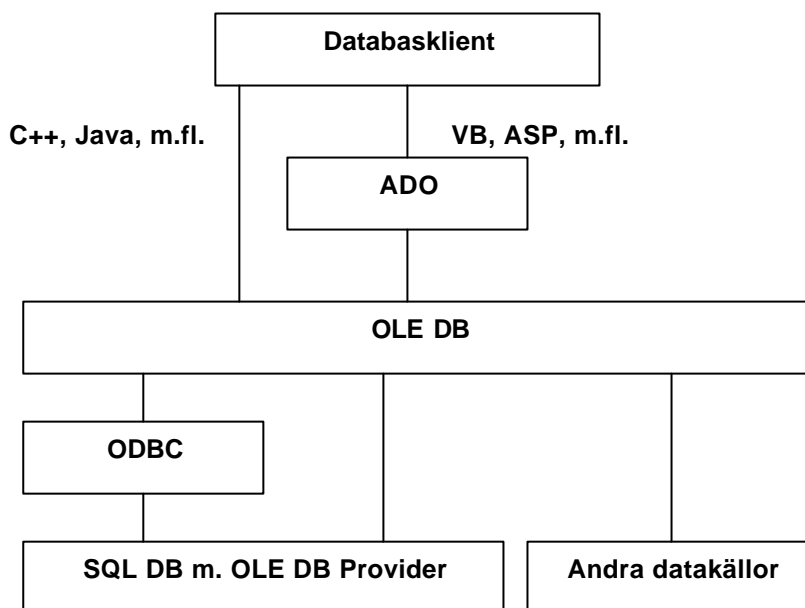
När vi jobbar med komponenter i Visual Basic så ärver **alla** komponenter ursprungligen från ett gränssnitt som heter `IUnknown`. Detta arv, och eventuella andra arv för komponentens funktion, sker ofta automatiskt (implicit). Andra tillfällen då arv kan bli aktuellt är när vi gör komponenter som använder Microsoft Transaction Server (MTS eller COM+) (se sammanfattning om DCOM/MTS).

3 Databasprogrammering med ADO

Microsoft ActiveX Data Objects (ADO) är den modell för åtkomst till databaser som Microsoft förordar för databasprogrammering i t.ex. Visual Basic (VB) och Active Server Pages (ASP). Modellen ersätter den enkla modellen Data Access Objects (DAO, som bygger på MS Jet/Access) och Remote Data Objects (RDO). RDO levererades med VB Enterprise Edition och var avsedd för snabbare åtkomst av databaser än vad DAO gör samt för åtkomst av databaser på andra datorer (vilket DAO inte klarar av).

Tanken med ADO är bl.a. att skapa en enhetlig modell, Universal Data Access (UDA), för åtkomst av datakällor (*data sources*). Med datakällor avses inte bara relationsdatabaser utan även "vanliga" filer,

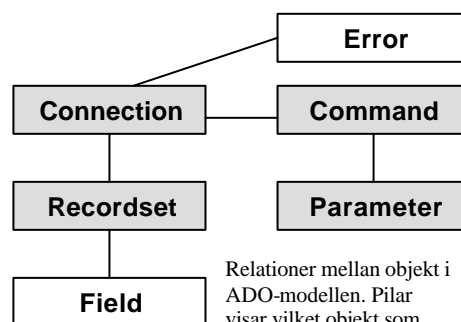
katalogtjänster¹², e-postsystem¹³, databaser som finns i stor-/minidatormiljö, m.m.. ADO använder sig av en annan teknik som heter OLE DB, som hanterar anslutning till databas samt åtkomst av data. OLE DB kan inte användas direkt i Visual Basic utan måste användas via ADO (vilket visserligen förenklar hanteringen av databaser). I bl.a. Visual C++ är det däremot lättare att använda OLE DB då det finns guider (*wizards*) för OLE DB men inte ADO.



Det ska nämnas att det finns två sätt att använda ADO-objekten. Antingen kan man sätta egenskaperna för respektive objekt först och sen kalla metoden **.Open()** eller så kan man skicka alla egenskaperna som parametrar till metoden **.Open()**. Open-metodens parametrar är frivilliga (*optional*), d.v.s. bara de egenskaper som önskas för objektet behöver skickas med. (Detta leder till att det egentligen finns ett tredje sätt att använda ADO-objekten: ange några egenskaper för ADO-objektet och skicka resten som parametrar till Open-metoden. Men vem räknar? ☺) Vi tittar närmare på hur de två sätten används i kod i nästa kapitel.

3.1 ADO-modellen

ADO-modellen består främst¹⁴ av fyra intressanta objekt: **Connection**, **Recordset**, **Command** och **Parameter**. Dessa objekt är grå markerade i figuren till höger och pilarna visar vilka objekt som refererar till vilka. T.ex. så behöver ett Recordset-objekt ett Connection-objekt för att vet vilken datakälla som används och ett Connection-objekt håller reda på felen som uppstår i en vektor med Error-objekt. Objekten Command och Parameter används för parametriserade frågor.



Relationer mellan objekt i ADO-modellen. Pilar visar vilket objekt som refererar till vilket.

¹² En katalogtjänst är databas som bygger på standarden X.500 och innehåller information om personer/användare, datorer, m.m.. Exempel på katalogtjänster är LDAP, Novells NDS och Microsofts AD.

¹³ Med e-postsystem (*message store*) så menas t.ex. MS Exchange.

¹⁴ Med främst menas ur Visual Basic-programmerarens synvinkel.

Ett Connection-objekt har till uppgift att hantera anslutningen (sessionen) till databasen under tiden som programmet läser och manipulerar poster i databasen. Vilken datakälla och *data provider* som anslutningen avser anges i Connection-objektets egenskap **.ConnectionString** (mer om detta senare). En *data provider* kan liknas med en drivrutin för datakälla.

Ett objekt av typen Recordset hanterar posterna i en tabellen - hela tabellen eller de poster som är resultatet av en fråga. För att hitta tabellerna i databasen måste egenskapen **.ActiveConnection** sättas att referera till ett objekt av typen Connection. Recordset-objektet vet vilken tabell eller frågeresultat den ska hantera genom att man sätter egenskapen **.Source** till tabellens namn eller en frågesträng (*query string*). Ett Recordset-objekt innehåller en vektor (*collection*) som heter **.Fields** som innehåller fälten i aktuell post, d.v.s. posten som objektet just nu pekar på. Värdena på fälten i posten kan (självklart!) nås på ett antal olika sätt (se *Egenskapen Fields* nedan).

3.1.1 Ett första exempel

Principen bakom databasprogrammering i Visual Basic är att:

1. öppna en anslutning till databasen – använda ett Connection-objekt.
2. öppna en tabell eller ställa en fråga mot databasen – använda ett Recordset-objekt.
3. manipulera poster i tabellen eller resultatet från frågan – m.h.a. Recordset-objekt.
4. eventuellt uppdatera posterna om de ändrats – m.h.a. Recordset-objekt.
5. stänga tabellen och anslutningen till databasen.

I detta exempel öppnas en anslutning (via MS Jet) till en Microsoft Access 2000-databas (MS Jet v4.0) som finns i roten på enhet C (C:\). Därefter öppnas tabellen `tblPersonal` och posterna loopas över för att visa tabellens alla fält `Id` i en listruta. Sist av allt stängs tabellen och anslutningen samt objekten förstörs, d.v.s. sätts till inget (`Nothing`).

För att prova koden kan ni skapa ett nytt projekt och sätta en referens till **Microsoft ActiveX Data Objects 2.x Library**¹⁵ genom att välja **References...** från Project-menyn. Placera sen en listruta med namnet `List1` på formuläret, dubbelklicka på formulärets bakgrund och fyll i nedanstående kod i proceduren `Form_Load()`.

3.1.1.1 Exempel:

```
'Deklarera variabler
Dim adoConn As ADODB.Connection
Dim adoRS As ADODB.Recordset

Set adoConn = New ADODB.Connection      'Skapa Connection-objekt

'Ange egenskaper för Connection-objektet
adoConn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" _
    & "Data Source=C:\bokning.mdb "

adoConn.Open                            'Öppna förbindelse

Set adoRS = New ADODB.Recordset         'Skapa Recordset-objekt

'Ange egenskaper för Recordset-objektet
Set adoRS.ActiveConnection = adoConn   'OBS! Set måste användas då objekt
adoRS.Source = "tblPersonal"          'Ange namn på tabell
adoRS.Open                             'Öppna tabellen

'Visa poster i en listruta (med namnet List1)
While Not adoRS.EOF                    'Så länge inte slut på poster..
    List1.AddItem adoRS!Id             '... lägg till fältet Id i listruta
    adoRS.MoveNext                    'Flytta till nästa post
Wend
```

¹⁵ Version 2.5 eller senare av ADO bör användas.

adoRS.Close	'Stäng förbindelser
adoConn.Close	
Set adoRS = Nothing	'Städa upp objekt
Set adoConn = Nothing	

3.2 Objektet Connection

Connection-objekt motsvarar en anslutning (förbindelse eller session) mot en datakälla. Generellt sätt så bör man bara öppna en förbindelse till en datakälla, d.v.s. variabler av typen Connection bör vara globala så att förbindelse mot datakälla endast behöver etableras en gång. Den enda gången som vi **inte** bör göra detta är i distribuerade komponenter – där måste anslutning etableras i varje metदानrop.¹⁶

Användbara egenskaper och metoder i objekt av typen Connection är:

- .ConnectionString
- .Open()
- .Execute()
- .Close()

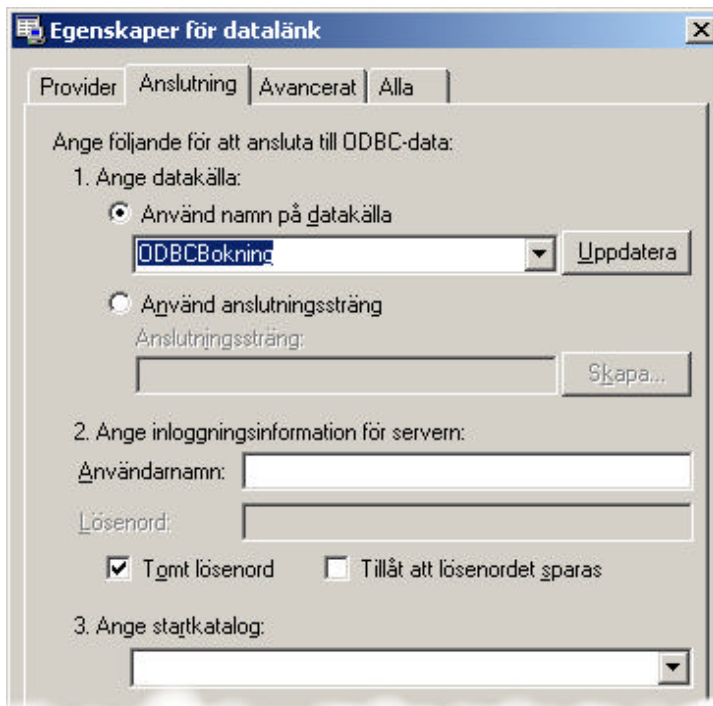
Även metoderna .BeginTrans(), .CommitTrans() och .RollbackTrans() är av intresse om vi själva vill hantera transaktioner (vilket vi inte vill med komponenter i MTS/COM+ ☺).

3.2.1 Egenskapen ConnectString

Egenskapen **.ConnectionString** i objektet Connection är, som nämnt tidigare, det som talar om vilken datakälla som ska anslutas till. I strängen anges vanligen dataleverantör (*provider* eller drivrutin) och datakälla (*data source*).

Men även egenskaper som användaridentitet, lösenord, hur dataåtkomst av andra ska hanteras (låsning) samt fler (för datakällan unika egenskaper) kan anges.

Lättaste sättet att skapa en sträng för anslutning är att skapa en fil med ändelsen .UDL (t.ex. DATA.UDL) och sen dubbelklicka på filen för att öppna dialogrutan Data Link Properties. Som standard är ODBC vald som *provider*. Vill man ändra detta så klickar man på fliken **Provider** och väljer en annan. Sen klickar man på **Nästa>/Next>**-knappen (eller klickar på fliken **Connection**) för att välja datakälla. Innehållet på fliken Connections varierar beroende på vilken typ av *provider* som valts. Till höger visas exempel för ODBC (figur 1), nedan Jet v4.0 (figur 2) och längst ner för Oracles egen drivrutin (figur 3).

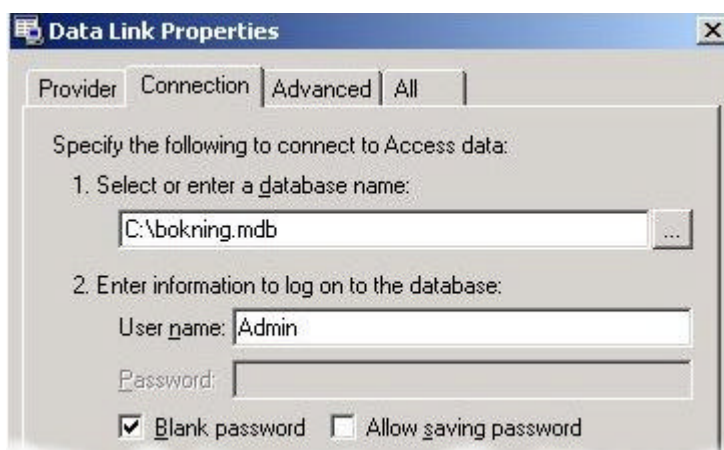


Figur 1 - Egenskaper för Connection med ODBC.

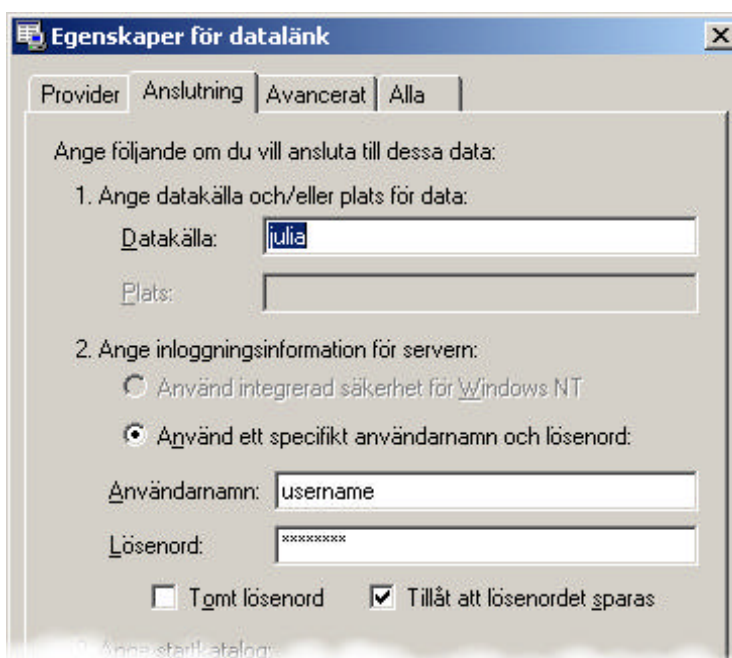
¹⁶ Distribuerade komponenter som installeras i Microsoft Transaction Server (MTS) får inte behålla sitt tillstånd mellan metदानrop (se sammanfattningen *Komponenter med DCOM/MTS*).

När man sen klickat på OK så sparas en sträng i filen (t.ex. DATA.UDL) som kan användas till **.ConnectionString**. Öppna filen med t.ex. Anteckningar och kopiera raden som börjar med Provider=. Man kan även hänvisa till filen vid anrop av **.Open()** (se exempel nedan).

Om vi använder databaser som kräver lösenord (till skillnad mot t.ex. Access) så kan vi fylla i användarnamn (*user ID*) och lösenord samt kryssa i kryssrutan *Tillåt att lösenord sparas* för att användarnamn och lösenord ska sparas som en del i **ConnectionString**.



Figur 2 – Egenskaper för Connection med Jet v. 4.0.



Figur 3 - Egenskaper för Connection med Oracles egen drivrutin.

En **ConnectionString** deklarerars lämpligen som en global konstant i VB-projektet, t.ex. längst upp i formulär eller publikt i en modul. På detta sätt så blir det relativt enkelt att byta (eller flytta) datakälla samt antagligen lättare att läsa databaskoden (d.v.s. när man öppnar förbindelse till datakälla).

3.2.1.1 Microsoft Access 2000/XP (Jet v4.0):

Om vi använder Access 2000/XP (egentligen *Jet Engine*) som datakälla så bör vi använda Jet som *data provider* (d.v.s. drivrutin). Nedan visas hur en **ConnectionString** för Jet version 4.0 ser ut. Databasfilen (.MDB-filen) finns i roten (C\) på enhet (hårddisk) C.

```
Dim adoConn As New ADODB.Connection
adoConn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" _
    & "Data Source=C:\bokning.mdb"
```

```
adoConn.Open
```

Använder vi en äldre version av Access (Jet), t.ex. Access 97, så ändrar vi 4.0 till 3.51 för egenskapen `Provider`.

3.2.1.2 ODBC (ADO 2.5+):

ODBC är användbart främst för datakällor som saknar OLE DB-drivrutiner (d.v.s. en *data provider*). Finns en *data provider* så bör (ska) vi använda den framför ODBC!

```
Dim adoConn As New ADODB.Connection
adoConn.ConnectionString = "Provider=MSDASQL.1;" _
    & "Data Source=ODBCBokning"
adoConn.Open
```

Eftersom ODBC är standarddrivrutinen kan därför `Provider` utelämnas och koden ovan kan skrivas om som:

```
Dim adoConn As New ADODB.Connection
adoConn.ConnectionString = "Data Source=ODBCBokning"
adoConn.Open
```

(Om vi använder ODBC med version 2.0 av ADO så saknas `.1` efter `MSDASQL` för egenskapen `Provider`.)

3.2.1.3 Oracle (med Oracles egna drivrutin):

På fliken `Provider` väljer vi *Oracle Provider for OLE DB* (om den finns) för att använda Oracles egna drivrutin.

```
Dim adoConn As New ADODB.Connection
adoConn.ConnectionString = "Provider=OraOLEDB.Oracle.1;Password=password;" _
    & "User ID=username;Data Source=julia"
adoConn.Open
```

Viktigt här är att vi fyller i användarnamn (*User ID*) och lösenord (*Password*) på fliken `Anslutning` om vi vill ha det som en del av vår `ConnectionString`. Alternativt kan vi skicka det som argument till metoden `.Open()` (se *Metoden Open* nedan).

3.2.1.4 Oracle (med Microsofts drivrutin):

På fliken `Provider` väljer vi *Microsoft OLE DB Provider for Oracle* för att använda Microsofts drivrutin för Oracle.

```
Dim adoConn As New ADODB.Connection
adoConn.ConnectionString = "Provider=MSDAORA.1;Password=password;" _
    & "User ID=username;Data Source=julia"
adoConn.Open
```

Viktigt här är att vi fyller i användarnamn (*User ID*) och lösenord (*Password*) på fliken Anslutning om vi vill ha det som en del av vår `ConnectionString`. Alternativt kan vi skicka det som argument till metoden `.Open()` (se *Metoden Open* nedan).

3.2.1.5 Använda .UDL-fil:

UDL-filer är praktiska att använda då dessa inte kompileras in i EXE-/DLL-filer, d.v.s. de kan ändras utan att behöva kompilera om EXE-/DLL-fil. En UDL-fil innehåller främst en `ConnectionString`.

```
Dim adoConn As New ADODB.Connection
adoConn.ConnectionString "File Name=C:\data.udl"
adoConn.Open
```

Observera dock att eventuell användaridentitet och lösenord kommer att spara i klartext i UDL-filen!

3.2.2 Metoden Open

Metoden `.Open()` öppnar förbindelsen till en datakälla och den har fyra parametrar: sträng med anslutningsegenskaper (`ConnectionString`), användaridentitet, lösenord och ett alternativ om när metoden ska avslutas. Anslutningssträngen har diskuterats i avsnittet *Egenskapen ConnectString* ovan. Användaridentitet och lösenord kan skickas med som separata parametrar eller ingå i anslutningssträngen. Om dessa parametrar skickas separat så kan man vid t.ex. inloggning fråga användaren efter dem. Den sista egenskapen är mindre frekvent använd och avgör om metod ska avslutas efter (synkront) eller innan (asynkront) förbindelse upprättats. Standard är att metoden avslutas efter att förbindelse har upprättats så att inte efterkommande metदानrop i t.ex. ett `Recordset`-objekt misslyckas. Om egenskapen `.ConnectString` har satts före anrop så är alla parametrar frivilliga

Det första exemplet nedan visar på hur egenskapen `.ConnectString` först sätts och sen att `.Open()` anropas. Som datakälla i de flesta exempel nedan har en ODBC-källa valts för att göra koden så kort som möjligt.

```
Dim adoConn As New ADODB.Connection
adoConn.ConnectionString = "Provider=MSDASQL.1; Data Source=ODBCBokning"
adoConn.Open 'ConnectionString satt => inga parametrar behövs
```

I nästa exempel skickas anslutningssträngen som parameter till `.Open()`. Genom att först tilldela `ConnectionString` till en variabel och sen använda variabeln som argument till metoden så blir förhoppningsvis koden lättare att läsa. (`ConnectionString` bör även deklarerars globalt enligt beskrivning ovan – se *Egenskapen ConnectString*).

```
Dim adoConn As New ADODB.Connection
Dim strConn As String
strConn = "Provider=MSDASQL.1;Data Source=ODBCBokning"
adoConn.Open strConn
```

Om en datakälla används i ett enanvändarsystem så kan förbindelse till datakällan t.ex. öppnas i metoden `Form_Load()` och sen stängas i `Form_Unload()` (förutsatt att datakällan inte används av flera användare). Variabel av typen `Connection` deklarerars då globalt.

När datakällor används av många, t.ex. i distribuerade komponenter, är det viktigt att öppna en förbindelse till datakällan så sent som möjligt, d.v.s. först när datakällan ska användas. Detta för att inte binda upp värdefulla resurser – just antalet förbindelser till datakällor är begränsade men även kostsamma (tidskrävande) att skapa. Variabler av typen Connection deklarerar då lokalt, d.v.s. i respektive metod.

3.2.3 Metoden Execute

Med hjälp av metoden **.Execute()** kan t.ex. en SQL-sats utföras mot Connection-objektets datakälla. Metoden kan fungera både som en procedur (returnerar inget) eller som funktion som returnerar ett objekt av typen Recordset.¹⁷ Metoden har tre parametrar, varav endast den första är obligatorisk. Parametrarna är en kommandosträng (t.ex. SQL-sats), en variabel (av typen Long) för att returnera antal påverkade poster och alternativ (av typen Long). Nedan visas ett exempel på hur **.Execute()** används för att öppna tabellen tblPersonal med hjälp av en SQL-fråga. Observera att resultatet av metoden är ett objekt av typen Recordset och därmed måste Set-operatören användas.

```
Dim adoConn AS New ADODB.Connection
Dim adoRS As ADODB.Recordset
Dim strConn As String, strQuery As String

strConn = "Provider=MSDASQL.1;Data Source=ODBCBokning"
adoConn.Open strConn
strQuery = "SELECT * FROM tblPersonal"
Set adoRS = adoConn.Execute(strQuery)
...
```

Om frågan är en uppdateringsfråga, som ändrar eller lägger till en post, så behövs det ingen mottagande variabel. Däremot kan man skicka med en variabel som argument (lngNewRow i exempel) för att ta reda på hur många poster som påverkades av kommandosträngen i första argumentet.

```
...
strQuery = "INSERT INTO tblPersonal VALUES ('bpn', 'Björn', 'Persson', '3678')"
adoConn.Execute strQuery, lngNewRows
MsgBox "Antal påverkade poster: " & lngNewRows
...
```

Tredje parametern till metoden **.Execute()** används för att tala om vilken typ av kommando som skickats i första parametern.

3.2.3.1 Konstanter för typer av kommando i kommandosträngar

Som tredje parameter till metoden **.Execute()** (och andra metoder i andra objekt) kan vi skicka fördefinierade konstanter. Några av konstanterna beskrivs i tabellen nedan.

Konstant	Beskrivning av kommandosträngs betydelse
----------	--

¹⁷ En procedur anropas genom att ge namnet på metoden, ett mellanslag och en kommaseparerad argumentlista (t.ex. adoConn.Execute "tblPersonal", lngAntal, adCmdTable). Vid anrop av funktioner däremot så innesluts argumentlistan inom parenteser och en variabel behövs för att hantera resultatet (t.ex. Set adoRS = adoConn.Execute("tblPersonal", lngAntal, adCmdTable)).

adCmdText	Kommando är av typen text (t.ex. en SQL-sats).
adCmdTable	Namn på tabell att öppna. <i>Data provider</i> konverterar tabellnamn till t.ex. en SQL-sats av typen <code>SELECT * FROM tabellnamn</code> .
adCmdTableDirect	Namn på tabell att öppna. Öppnar tabellen direkt utan att konvertera till t.ex. SQL-sats.
adCmdStoredProc	Lagrad procedur (<i>stored procedure</i>) att utföra i datakälla.
adCmdUnknown	Typ av kommando är okänt – datakälla får (försöka) avgöra typ av kommando själv. Detta är inte ett lämpligt alternativ men är standard för tredje parameter i metoden <code>.Execute()</code> .
adExecuteNoRecords	Kommando ska utföras utan att returnera några poster. Denna konstant används främst i kombination (genom att addera med annan konstant) för att effektivisera utförandet av kommando, t.ex. om SQL-sats innehåller en UPDATE-sats.

3.2.4 Metoden Close

Metoden `.Close()` stänger förbindelsen till datakällan och har inga parametrar. I VB sker detta automatiskt när variabeln förstörs (d.v.s. när metoden avslutas) men det är god sed att explicit stänga förbindelsen. När datakällor används i komponenter är det lika viktigt att stänga förbindelsen så fort som möjligt som det är att öppna den så sent som möjligt.

```
adoConn.Close
```

3.3 Objektet Recordset

Recordset-objekt motsvarar poster i en tabell (eller frågeresultat) och det mesta av manipulerandet av databaser sker med hjälp av dessa objekt. Hur manipulationen av poster kan ske beror på värden vi ger Recordset-objektets egenskaper – egenskaper som oftast måste anges **innan** objektet öppnas (metoden `.Open()` anropas). Innan vi kan använda ett Recordset-objekt bör vi ha skapat ett Connection-objekt (se ovan).

Variabler av typ Recordset bör deklarerars med så liten räckvidd som möjligt, d.v.s. helst lokalt i metoder (om objektet inte hålls öppet och används av flera metoder).

Några av de mest användbara egenskaperna och metoderna är:

- `.ActiveConnection`
- `.Source`
- `.CursorType`
- `.LockType`
- `.Fields`
- `.AddNew()`
- `.BOF()` och `.EOF()`
- `.Close()`
- `.Move()`, `.MoveFirst()`, `.MovePrevious()`, `.MoveNext()` och `.MoveLast()`
- `.Open()`
- `.Update()`

3.3.1 Egenskapen ActiveConnection

Som nämnt tidigare så krävs en referens till datakällan (genom ett Connection-objekt) för att ett Recordset-objekt ska veta vilken datakälla som tabeller finns i. Egenskapen

.ActiveConnection hanterar referensen med Connection-objektet. Eftersom ett Connection-objekt är just ett objekt så måste **.ActiveConnection** sättas att referera till Connection-objektet med Set-operatoren:

```
Dim adoConn As New ADODB.Connection
Dim adoRS As New ADODB.Recordset

adoConn.Open "Data Source=ODBCBokning"
Set adoRS.ActiveConnection = adoConn      'adoConn är ett objekt - anv. Set!
adoRS.Open "tblPersonal"
```

Connection-objektet kan även skickas som parameter till **.Open()** (vilket främst är det sätt att ange värden på **.ActiveConnection** som kommer användas i resterande exempel i denna sammanfattning).

```
...
adoConn.Open "Data Source=ODBCBokning"
adoRS.Open "tblPersonal", adoConn      'Skicka ActiveConnection som parameter
```

3.3.2 Egenskapen Source

I objektet Recordset är det egenskapen **.Source** som anger vilka poster objektet hanterar. **.Source** är ett kommando (en sträng) som t.ex. kan vara namnet på en tabell eller en SQL-sats.

- ① Denna egenskap måste sättas antingen innan anrop av, eller skickas som parameter till, metoden **.Open()**. Undantag är om metoden **.Execute()** i ett Connection-objekt används för att skapa Recordset-objektet (se avsnittet *Objektet Connection* ovan).

Nedan visas fyra exempel på hur egenskapen **.Source** kan anges.

```
adoRS.Source = "tblPersonal"
adoRS.Source = "SELECT * FROM tblPersonal"      'samma resultat som ovan
adoRS.Source = "SELECT * FROM tblPersonal WHERE Id LIKE 'B%'"
adoRS.Open "tblPersonal", adoConn
```

3.3.3 Egenskapen CursorType

Uppgiften för ett Recordset-objekt är att hanterar posterna i tabellen (eller frågeresultatet). För att manipulera data i poster så görs detta på post per post basis, d.v.s. en post åt gången (se *Egenskapen Fields* nedan). För att kunna komma åt de enskilda posterna i tabellen används en postpekare (*cursor*) för att peka på aktuell post (den vars data kan manipuleras). Pekaren kan flyttas m.h.a. metoderna **.MoveFirst()**, **.MoveLast()**, **.MovePrevious()**, **.MoveNext()** och **.Move()**.

Hur postpekaren kan flyttas avgörs av egenskapen **.CursorType**. Desto mer flexibel pekare är (mer funktionalitet som stöds av pekare), desto mer kostsamt är det att manipulera posterna. Vi bör därför välja typ av postpekare efter typ av funktionalitet som krävs. D.v.s. behöver vi endast loopa genom posterna för att skriva i en textruta så bör vi öppna tabellen med den enklaste postpekaren (`adOpenForwardOnly`). I tabellen nedan visas konstanter för de vanligaste sätten på hur en postpekare kan flyttas. Konstanterna har ordnats i minst krävande (minst funktionalitet) till mest krävande (mest funktionalitet).

Konstant	Förklaring
----------	------------

adOpenForwardOnly	Pekare kan endast flyttas framåt (till nästa post). Inga förändringar i datakälla visas (t.ex. nya eller raderade poster). Detta är standard om inte värde på denna egenskap anges.
adOpenStatic	Pekaren kan flyttas både framåt och bakåt bland posterna. Inga förändringar i tabell visas (t.ex. nya eller raderade poster).
adOpenKeyset	Pekaren kan flyttas både framåt och bakåt bland posterna. Alla förändringar i tabell visas utom nya poster (t.ex. uppdaterad och raderade poster).
adOpenDynamic	Pekaren kan flyttas både framåt och bakåt bland posterna. Alla förändringar i tabell visas (t.ex. nya, uppdaterad och raderade poster).

- ① Denna egenskap har ett standard värde (adOpenForwardOnly) och behöver endast sättas om annat än standardvärdet önskas. Observera att inte alla datakällor stödjer alla alternativ!

3.3.3.1 Exempel:

Denna egenskap kan även skickas som tredje argument till metoden **.Open()** (se *Metoden Open* nedan).

```
adoRS.CursorType = adOpenKeyset
adoRS.Open strQuery, adoConn, adOpenKeyset
```

3.3.4 Egenskapen LockType

För att garantera riktigheten hos data i en databas måste enstaka poster eller hela tabeller låsa när uppdatering av data sker. Vilken typ av låsning som önskas kan anges genom att sätta egenskapen **.LockType** i objektet Recordset till en av värdena i tabellen nedan.

Att välja låsningstyp påverkar prestanda och möjligheten till konflikter mellan data ändrade av olika användare i en fleranvändardatakälla.

Konstant	Förklaring
adLockReadOnly	Tabellen öppnas för enbart läsning (standard).
adLockOptimistic	Tabellen öppnas för både läsning och skrivning. Post låses när Update-metoden anropas.
adLockPessimistic	Tabellen öppnas för både läsning och skrivning. Post låses (oftast) så fort redigering av ett värde i posten sker till dess att Update-metoden anropas.
adLockBatchOptimistic	Tabellen öppnas för både läsning och skrivning. All poster som ändrats i ett Recordset uppdateras på en gång, d.v.s. när UpdateBatch-metoden anropas. Detta alternativ är främst användbart med <i>disconnected</i> Recordset-objekt.

Observera att inte alla datakällor stödjer alla låsningstyper. För att kontrollera om en datakälla stödjer låsningstypen kan man anropa metoden **.Supports()** i ett Recordset-objekt.

```
Dim adoRS As ADO.DB.Recordset
Dim blnSupports As Boolean
...
' Stödjer tabellen pessimistisk låsning?
blnSupports = adoRS.Supports(adLockPessimistic)
...
```


- ① Denna egenskap har ett standard värde (`adReadOnly`) och behöver endast sättas om annat än standardvärdet önskas.

3.3.5 Egenskapen **Fields**

Som nämnts ovan så innehåller objektet `Recordset` en egenskap **Fields** i form av en vektor (av typen `Collection`). Vektorn innehåller "alla poster" i tabell eller frågeresultat – när vi använder denna vektor så är det egentligen fälten i den aktuella posten som postpekare pekar på. För att flytta till annan post så använder vi någon av `Move`-metoderna (se nedan).

Det finns tre sätt för åtkomst av fältens värden i vektorn **Fields** (d.v.s. aktuell post):

1. Direkt via fältets namn (`!FältNamn`).
2. Genom att indexera i vektorn med fältets namn som nyckel (`.Fields("FältNamn")`).
3. Genom att indexera i vektorn med indextal (`.Fields(2)`). Vektorns index börjar på 0!

Exemplet nedan visar hur vi hämtar värden från en post med alla tre sätt.

```
While Not adoRS.EOF                                'Så länge inte slut på poster..
    'Bygg sträng att lägga till i listruta
    strPerson = adoRS!Id & " " _
                & adoRS.Fields("FNamn") & " " _
                & adoRS.Fields(2)
    List1.AddItem strPerson
    adoRS.MoveNext
Wend
'Id-fält
'FNamn-fält
'ENamn-fält
'Lägg till sträng till listruta
'Flytta till nästa post
```

- ① Värde för denna egenskap (vektorn) kan inte sättas (d.v.s. en ny post skapas manuellt – använd metoden **AddNew()** – se *Metoden AddNew* nedan). Däremot kan värden sättas för varje medlem i vektorn, d.v.s. respektive fält i aktuell post.

```
...
adoRS.addNew                                       'Skapa ny post (d.v.s. en ny vektor)
adoRS!Id = "bpn"                                  'Sätt värden (i vektorn)
adoRS.Fields("FNamn") = "Björn"
adoRS.Fields(2) = "Persson"
adoRS.Update                                       'Spara post
```

Denna vektor innehåller alla fält i aktuell post och faktum är att dessa fält objekt är av typen `Field`. Men när vi använder egenskapen **Fields** så kan vi i de flesta fall strunta i detta faktum. Men i vissa fall, t.ex. när vi tilldelar till variabler och implicit konvertering inte fungerar, så måste vi använda egenskapen `Value` i `Field`-objekt för att hämta värdet på fältet och inte själva fältet (d.v.s. `Field`-objektet). Ett exempel är om vår variabel är av typen `Variant` (som alla variabler är i t.ex. ASP).

```
Dim varObj As Variant
varObj = adoRS.Fields("FNamn").Value 'Hämta värdet och inte hela Field-objektet
```

3.3.6 Metoden AddNew

.AddNew() används för att lägga till en ny post i Recordset-objektet. När post skapats kan vi använda egenskapen **.Fields** för att sätta värden på respektive fält i den nya posten. Sist av allt måste vi anropa metoden **.Update()** för att spara den nya posten.

```
Set adoRS = New ADODB.Recordset           'Skapa Recordset-objekt

With adoRS
  .Open strSQL, adoConn, , adLockOptimistic 'Öppna tabell med låsning
  .AddNew                                   'Lägg till ny post
  .Fields("Id") = txtAnvID                 'Sätt värden på ny posts fält
  .Fields("FNamn") = txtFornamn
  .Fields("ENamn") = txtEfternamn
  .Fields("Telefon") = txtTelefon
  .Update                                   'Spara ny post
  .Close                                    'Stäng tabell
End With
```

I exempel ovan har **With**-konstruktionen (**With ... End With**) används för att slippa behöva upprepa variabeln **adoRS** först i varje programsats. Om vi skriver en punkt först så avses variabeln som står efter det reserverade ordet **With**.

3.3.7 Metoderna BOF och EOF

Metoderna **.BOF** (Beginning-Of-File) och **.EOF** (End-Of-File) används, vid t.ex. loopar över poster, för att kontrollera om man kommit till början resp. slutet av poster. Ett typiskt exempel är:

```
While Not adoRS.EOF                       'Så länge inte slut på poster..
  strPerson = adoRS!Id & " " _
              & adoRS!FNamn & " " _
              & adoRS!ENamn               'Id-fält
                                          'FNamn-fält
                                          'ENamn-fält
  List1.AddItem strPerson                 'Lägg till sträng till listruta
  adoRS.MoveNext                           'Flytta till nästa post
Wend
```

I detta exempel loopas det över posterna och tre fält (**Id**, **FNamn** och **ENamn**) läggs till i en listruta med mellanslag mellan fälten.

Ett sätt att testa om en frågesträng gav en eller fler poster som resultat är att använda både **.BOF** och **.EOF**, d.v.s. att tabellpekaren inte står i både börja och slutet:

```
If adoRS.BOF AND adoRS.EOF Then
  MsgBox "Frågans resultat innehåller inga poster"
Else
  'Gör något med posterna
  ...
End If
```

3.3.8 Metoden Close

Som namnet på metoden visar så stänger **.Close()** den öppna tabellen. Metoden har inga parametrar.

```
adoRS.Close
```

- ① Denna metod anropas implicit då variabeln som refererar till Recordset-objektet inte är giltig längre (d.v.s. i slutet på t.ex. en metod). Men det är god sed att alltid anropa **.Close()** explicit i slutet på metoden eller då Recordset-objektet inte behövs längre. Glöm inte att stänga öppna anslutningar och tabeller så fort som möjligt då databaser används i komponenter.

3.3.9 Metoderna Move, MoveFirst, MovePrevious, MoveNext och MoveLast

Metoderna används för att flyttar postpekare (aktuell position) i Recordset-objektet. Postpekaren används för att peka på aktuell post, d.v.s. post som kan manipuleras.¹⁸ Metoden **.Move(index)** flyttar till postnummer motsvarande `index`. Övriga metoder flyttar till första, föregående, nästa resp. sista posten i Recordset-objektet.

Den mest frekvent använda metoden är **.MoveNext()** som flyttar till nästa post. Vi kan t.ex. använda denna metod när vi loopar över alla poster för utskrift. (Övriga metoder används inte så ofta i komponenter.)

```
adoRS.Move(4)
```

I exemplet blir aktuell post nummer 4.

- ① Observera att typen av postpekare (se *Egenskapen CursorType* ovan) avgör hur postpekare kan flyttas – vissa postpekare tillåter t.ex. endast att pekare flyttas till nästa post.

3.3.10 Metoden Open

Metoden **.Open()** öppnar en tabell eller resultat av en fråga (mot en eller flera tabeller i datakällan) och har fem parametrar: kommandosträng (**.Source** – se ovan), Connection-objekt (**.ActiveConnection** – se ovan), typ av pekare (**.CursorType** – se ovan), typ av låsning (**.LockType** – se ovan) och typ av kommando i första parametern. Den sista parametern är inte obligatorisk och kan vara t.ex. `adCmdTable` eller `adCmdTableDirect` (se *Metoden Execute* i Connection-objektet för konstanter).

```
Dim adoConn As New ADODB.Connection
Dim adoRS As New ADODB.Recordset

adoConn.Open "Data Source=ODBCBokning"
adoRS.Open "tblPersonal", adoConn, adOpenKeyset, adLockOptimistic, adCmdTable
```

- ① Parametrarna för pekare, låsning och typen av kommando (parametrar tre, fyra respektive fem) har standardvärden (`adOpenForwardOnly`, `adLockReadOnly` resp. `adCmdUnknown`).

3.3.11 Metoden Update

Sparar ändringar i Recordset-objekt, t.ex. ny post, uppdateringar av en post eller borttagning av en post. **.Update()** anropas automatiskt bl.a. om postpekare flyttas till en annan post med någon av Move-metoderna.

```
adoRS.Update
```

¹⁸ Endast en post åt gången kan manipuleras (t.ex. läsas eller uppdateras). Därför behövs postpekaren för att peka på aktuell post.

3.4 Objektet Command (och Parameter)

Command-objekt används bl.a. för att kunna köra frågor med parametrar ("parametriserade" frågor) men även för att kunna anropa lagrade procedurer (*stored procedures*) med parametrar och/eller andra returvärden än Recordset-objekt.

Parameter-objektet är endast intressant i samband med Command-objekt. Därför beskrivs detta objekt i samband med metoden **.CreateParameter()**.

Några av de mest användbara egenskaperna och metoderna är:

- .ActiveConnection
- .CommandText
- .CommandType
- .CreateParameter()
- .Execute()

3.4.1 Egenskapen ActiveConnection

Denna egenskap fungerar som den i Recordset-objektet (se *Egenskapen ActiveConnection* ovan).

3.4.2 Egenskapen CommandText

.CommandText används för att ange kommando att utföra – t.ex. en SQL-sats eller en lagrad procedur.

3.4.3 Egenskapen CommandType

.CommandType används för att tala om vilken typ av kommando som egenskapen **.CommandText** innehåller. (Dessa är samma som för parameter i Connection-objektets metod **Execute()**).

3.4.3.1 Konstanter för typer av kommando i kommandosträngar

Några av konstanterna beskrivs i tabellen nedan.

Konstant	Beskrivning av kommandosträngs betydelse
adCmdText	Kommando är av typen text (t.ex. en SQL-sats).
adCmdTable	Namn på tabell att öppna. <i>Data provider</i> konverterar tabellnamn till t.ex. en SQL-sats av typen <code>SELECT * FROM tabellnamn</code> .
adCmdTableDirect	Namn på tabell att öppna. Öppnar tabellen direkt utan att konvertera till t.ex. SQL-sats.
adCmdStoredProc	Lagrad procedur (<i>stored procedure</i>) att utföra i datakälla.
adCmdUnknown	Typ av kommando är okänt – datakälla får (försöka) avgöra typ av kommando själv. Detta är inte ett lämpligt alternativ men är standard för tredje parameter i metoden .Execute() .
adExecuteNoRecords	Kommando ska utföras utan att returnera några poster. Denna konstant används främst i kombination (genom att addera med annan konstant) för att effektivisera utförandet av kommando, t.ex. om SQL-sats innehåller en UPDATE-sats.

3.4.4 Egenskapen Parameters

Denna egenskap är en vektor (av typen Collection) som innehåller alla parametrar (både in- och utparametrar) som skickas till och tas emot från datakälla.

För att lägga till en parameter i egenskapen så använder vi metoden Add() (i egenskapen **.Parameters**). Lämpligen lägger vi till parametrarna i den ordning som dom förekommer i kommandot (som anges med egenskapen **.CommandText**).

```
adoCmd.Parameters.Add(adoParam)
```

För att läsa ett värde från en utparameter så kan vi använda en "genväg" och bara ange namnet på egenskapen samt index för parameter som vi vill hämta värdet från. I exempel nedan så hämtar vi värdet på första parametern (som t.ex. skulle kunna vara registreringsnumret på en bil).

```
strRegnr = adoCmd.Parameters(0)
```

3.4.5 Metoden CreateParameter()

.CreateParameter() används för att skapa parametrar som motsvarar variabla värden i ett kommando. Metoden har fem parametrar som alla är frivilliga (även om vi lämpligen använder någon av dem ☺).

- **Namn** – namn på parameter (och eventuellt även på parameter i lagrad procedur).
- **Typ** – på värde som skickas (t.ex. sträng [adwChar], datum [adDate] – sök på "Type property" och välj *Microsoft ADO Help* i MSDN).
- **Riktning** – in, ut eller både in och ut (eller returvärde från en lagrad funktion). Här används ofta konstanter, d.v.s. adParamInput, adParamOutput respektive adParamInputOutput (eller adParamReturnValue).
- **Storlek** – på värde som skickas. Används främst för strängar (t.ex. genom att använda funktionen Len()).
- **Värde** – som parameter representerar. Endast relevant för inparametrar.

I nedanstående exempel skapas en inparameter av typen sträng (adwChar¹⁹) där värdet finns i variabeln strRegnr. Eftersom ett registreringsnummer är (eller bör vara) en sträng så använder vi fjärde argumentet för att tala om längden på strängen.

```
adoParam = adoCmd.CreateParameter("Regnr", adwChar, adParamInput, Len(strRegnr), _  
strRegnr)
```

När vi skapat ett Parameter-objekt använder vi egenskapen **.Parameters** för att lägga till parametern till Command-objektet (se *Egenskapen Parameters* ovan).

(Se nästa kapitel för fler exempel.)

¹⁹ Strängar i VB består tecken av storleken 2 byte (som Unicode). Därför används konstanten adwChar för att visa att det är en *wide character*.

3.4.6 Metoden Execute()

Metoden **.Execute()** används för att utföra kommandot som angavs med egenskapen **.CommandText** eller som bifogas som argument till metod. I nedanstående utförs en SQL-sats (i variabeln `strQuery`) som returnerar ett antal poster (d.v.s. ett Resultset-objekt).

```
Set adoRS = adoCmd.Execute(strQuery) 'Utför en SQL-sats som returnerar Recordset
```

Returvärdet från metoden beror på typ av kommando. Om kommandot är en SELECT-sats så returneras t.ex. ett Recordset-objekt (som i exempel ovan). Men om det är en lagrad procedur (eller lagrad funktion) beror returvärdet på eventuellt returvärde och/eller om in-/utparametrar används i lagrad procedur. (Se nästa kapitel för exempel.)

3.5 Skapa obundna Recordset-objekt

En av de saker som ADO används för är att skapa obundna (*disconnected*) Recordset-objekt – objekt som kan skickas till klienten. Om klienten uppdaterar några poster i tabellen så behöver endast de uppdaterade posterna skickas tillbaka till servern, s.k. *batch update*. Fördelen med detta är att klienten öppnar en session mot databasen, hämtar postern och avslutar sessionen – klienten håller m.a.o. inte posterna låsta. Nackdelen är att en annan klient kan ha uppdaterat posterna under tiden, ett problem som måste hanteras (på ett eller annat sätt ☺).

För att skapa ett obundet Recordset-objekt måste vi ange att klienten (till datakälla!) ska hantera postpekaren samt koppla loss objektet från datakällan. Det första gör vi **innan** vi öppnar anslutningen till datakällan och det sista innan vi skickar iväg Recordset-objektet (t.ex. som ett returvärde från funktion).

```
Set adoConn = New ADO.Connection           'Anv. postpekare i klient  
adoConn.CursorLocation = adUseClient      'Öppna datakälla  
adoConn.Open strConn                      'Hämta poster  
Set adoRS = adoConn.Execute(strsql)      'Koppla loss från datakälla  
adoRS.ActiveConnection = Nothing
```

3.6 Konstruera SQL-satser i Visual Basic-kod

När vi jobbar med ADO (och andra databas-API) så måste vi oundvikligen vid något tillfälle konstruera ("bygga") SQL-satser. SQL-satserna konstrueras i VB (värdspråket) som en sträng (d.v.s. inneslutna inom citattecken – dubbla i VB). I nedanstående exempel tilldelas en (VB-)sträng, innehållande en SQL-sats där alla bilar med registreringsnumret ABC123 hämtas, till variabeln `strSQL`.

```
strSQL = "SELECT * FROM bilar WHERE regnr = 'ABC123' "
```

SQL-satsen ovan fungerar endast om vi alltid vill hämta bilar med registreringsnumret ABC123. Men ofta vill vi kunna be användaren ange t.ex. ett registreringsnummer (eller annat varierande värde). Detta kan vi t.ex. göra genom att sammanfoga strängen (med SQL-satsen) med variabler som innehåller de varierande värdena. (Detta är dock ett sätt som bör undvikas då VB-koden kan bli svårsläst samt att det ofta är svårare att felsöka! Det är bl.a. lätt att glömma de enkla citattecknen kring strängar i SQL-satsen.)

```
strRegnr = "ABC123"  
strSQL = "SELECT * FROM bilar WHERE regnr = '" & strRegnr & "'"
```

En bättre lösning är att använda ett Command-objekt och parametrar för de varierande värdena. Varje värde som kan variera markeras med ett frågetecken i SQL-sats och vi skapar ett Parameter-objekt för varje frågetecken i SQL-sats. I nedanstående exempel används en parameter (frågetecken) för att kunna hämta bilar med olika registreringsnummer.

```
strRegnr = "ABC123"  
strSQL = "SELECT * FROM bilar WHERE regnr = ?"  
'...  
adoParam = adoCmd.CreateParameter(, adWChar, adParamInput, Len(strRegnr), _  
                                strRegnr)
```

I ovanstående exempel skickas endast argument för parametrarna motsvarande typ för värde (adWChar – en "Unicode" sträng), riktning för parameter (adParamInput – en inparameter), längd på sträng (Len(strRegnr) – behövs bara om sträng) respektive själva värdet (strRegnr). Vi har alltså utelämnat namnet på parametern (eftersom det inte är relevant då kommandot är en SQL-sats).

(Denna sida har avsiktligt lämnats blank.)

4 Databasfunktioner i kod

I detta kapitel ska vi titta på mer sammanhängande kod för att få en helhet.

4.1 Öppna tabell och hämta poster

I detta exempel öppnas tabellen `tblPersonal`, som ODBC-källan `ODBCBokning` refererar till, och första fältet (`Id`) läggs till i en listruta. Vill man inte visa alla poster i tabellen kan man ersätta `tblPersonal` med en SQL-fråga istället (bort kommenterad i första exemplet).

Detta kan ske på främst två sätt:

- Skapa ett `Recordset`-objekt och använda dess metod `Open()`.
- Använda metoden `Execute()` i `Connection`-objekt.

Här visas även (sist) exempel på hur ett `Connection`-objekt kan skapas implicit.

4.1.1 Skapa Recordset-objekt och använda dess metod Open()

Om vi använder detta sätt att öppna en tabell så gör vi följande:

1. Skapa ett `Connection`-objekt, ange egenskaper och öppna förbindelse till datakälla.
2. Skapa ett `Recordset`-objekt, ange egenskaper och öppna tabell/kör fråga.
3. Manipulera poster.
4. Stäng tabell/frågeresultat och datakälla (samt städa upp).

```
Dim adoConn As ADODB.Connection 'Deklarera variabler
Dim adoRS As ADODB.Recordset
Dim strConn As String, strRSSource As String

'Skapa Connection-objekt, ange anslutningssträng och öppna anslutning
Set adoConn = New ADODB.Connection
strConn = "Provider=MSDASQL;Data Source=ODBCBokning"
adoConn.Open strConn

'Skapa Recordset-objekt och öppna tabellen med Connection-objekt
Set adoRS = New ADODB.Recordset
strRSSource = "tblPersonal"
' strRSSource = "SELECT * FROM tblPersonal WHERE Id LIKE 'b%'"
adoRS.Open strRSSource, adoConn

'Manipulera poster
While Not adoRS.EOF 'Så länge inte slut på poster..
    List1.AddItem adoRS!Id '.. lägg till fältet Id i listruta
    adoRS.MoveNext 'Flytta till nästa post - glöm INTE!!
Wend

adoRS.Close 'Stäng förbindelser
adoConn.Close
Set adoRS = Nothing 'Städa upp objekt
Set adoConn = Nothing
```

4.1.2 Använda metoden .Execute() i Connection-objekt

Om vi använder detta sätt att öppna en tabell så gör vi följande:

1. Skapa ett `Connection`-objekt, ange egenskaper och öppna förbindelse till datakälla.
2. Anropa metoden `Execute()` i `Connection`-objekt med tabell att öppna/fråga att köra som parameter. D.v.s. skapa inget (eget) `Recordset`-objekt.
3. Manipulera poster.

4. Stäng tabell/frågeresultat och datakällas (samt städa upp).

Steg 1, 3 och 4 är alltså de samma som i exempel ovan. Skillnaden (i steg 2) har markerats med grå bakgrund i exempel nedan.

```
Dim adoConn As ADODB.Connection 'Deklarera variabler
Dim adoRS As ADODB.Recordset
Dim strConn As String, strRSSource As String

'Skapa Connection-objekt, ange anslutningssträng och öppna anslutning
Set adoConn = New ADODB.Connection
strConn = "Provider=MSDASQL;Data Source=ODBCBokning"
adoConn.Open strConn

'Öppna tabellen med metoden Execute (istället för att öppna Recordset-objekt)
strRSSource = "SELECT * FROM tblPersonal"
Set adoRS = adoConn.Execute(strRSSource) 'Anropa Execute f. att skapa Recordset

'Manipulera poster
While Not adoRS.EOF 'Så länge inte slut på poster..
    List1.AddItem adoRS!Id '.. lägg till fältet Id i listruta
    adoRS.MoveNext 'Flytta till nästa post - glöm INTE!!
Wend

adoRS.Close 'Stäng förbindelser
adoConn.Close
Set adoRS = Nothing 'Städa upp objekt
Set adoConn = Nothing
```

Observera att vår SQL-sats **inte** avslutas med ett semikolon (;) – ibland kan t.o.m. fel uppstå om man använder det! Detta gäller i de flesta lägen som vi använder SQL-satser och de flesta typer av SQL-satser – inte bara när vi kör SELECT-satser.

4.1.3 (Skapa ett Connection-objekt implicit)

Vill man spara lite kod (☺) så behöver man inte explicit skapa ett Connection-objekt. Man kan låta Recordset-objektet implicit skapa referensen mot databasen. Koden i exemplet ovan skrivs om enligt följande (kod med grå bakgrund).

Om vi använder detta sätt att öppna en tabell så gör vi följande:

1. Skapa endast en ConnectString.
2. Skapa ett Recordset-objekt, ange egenskaper och öppna tabell/kör fråga.
3. Manipulera poster.
4. Stäng tabell/frågeresultat och datakällas (samt städa upp).

I detta exempel så är det endast steg 1 och 2 som har ändrats. Ändringen i steg 2 är att vi skickar en ConnectString istället för ett Connection-objekt som parameter två till Recordset-objektets metoden Open().

```
'Deklarera variabler
Dim adoRS As ADODB.Recordset
Dim strConn As String, strRSSource As String

'Skapa anslutningssträng (utan att öppna anslutning)
strConn = "Provider=MSDASQL;Data Source=ODBCBokning"

'Skapa Recordset-objekt och öppna tabellen
Set adoRS = New ADODB.Recordset
strRSSource = "tblPersonal"
adoRS.Open strRSSource, strConn 'Använd strConn istället för Connection-obj
```

```
'Manipulera poster
While Not adoRS.EOF
    List1.AddItem adoRS!Id
    adoRS.MoveNext
Wend

adoRS.Close
adoConn.Close
Set adoRS = Nothing
Set adoConn = Nothing

'Så länge inte slut på poster..
'.. lägg till fältet Id i listruta
'Flytta till nästa post - glöm INTE!!

'Stäng förbindelser
'Ståda upp objekt
```

Denna form av ”genväg”, att inte använda sig av ett Connection-objekt, rekommenderas **inte** om man öppnar två tabeller samtidigt. Då kommer nämligen två referenser till databaser (Connection-objekt) att implicit skapas. Och eftersom det är resurskrävande att skapa referenser mot databaser så rekommenderas inte detta.

4.2 Lägg till post i tabell

Här öppnas tabellen `tblPersonal`, en ny post skapas, fält i posten tilldelas värden samt sist sparas posten och tabellen stängs. När vi lägger till poster så kan detta ske på främst två sätt:

- Använda Recordset-objekt.
- Använda SQL-sats och metoden `Execute()` i Connection-objekt.

Om vi inte vill öppna hela tabellen så skulle vi kunna använda en SQL-sats som inte hämtar några poster, d.v.s. som har ett villkor som alltid kommer vara falskt (t.ex. ett ”ogiltigt”²⁰ värde på primärnyckeln).

4.2.1 Använda Recordset-objekt

Att lägga till poster skiljer sig inte så mycket från att hämta poster från en datakälla. Skillnaden ligger i manipuleringen av poster (steg 3 i exempel ovan).

Observera den extra parametern till metoden **.Open()** som behövdes för att kunna låsa tabellen innan uppdatering. I detta fall använde jag konstanten `adLockOptimistic` för att endast låsa den nya posten vid uppdatering av tabellen (se vidare under *Egenskapen LockType* i föregående kapitel). I exemplet nedan används `With`-satsen för att variabeln `adoRS` inte ska behöva upprepas framför varje metod.

```
Dim adoConn As ADODB.Connection 'Deklarera variabler
Dim adoRS As ADODB.Recordset
Dim strConn As String, strRSSource As String

'Skapa Connection-objekt, ange anslutningssträng och öppna anslutning
Set adoConn = New ADODB.Connection
strConn = "Provider=MSDASQL;Data Source=ODBCBokning"
adoConn.Open strConn

'Skapa Recordset-objekt och öppna tabellen med låsning för skrivning
Set adoRS = New ADODB.Recordset
strRSSource = "tblPersonal"
adoRS.Open "tblPersonal", adoConn, , adLockOptimistic, adCmdTable

'Manipulera poster - använd With för att slippa behöva upprepa adoRS
With adoRS
    .AddNew 'Lägg till ny post
    !Id = Text1(0) 'Sätt fältet Id i nya posten
    !FNamn = Text1(1) 'Sätt fältet FNamn i nya posten
```

²⁰ Med ”ogiltigt” menas ett värde som aldrig kan förekomma, t.ex. -1 för ordernummer (bör vara positivt ☺).

```
!ENamn = Text1(2)      'Sätt fältet ENamn i nya posten
!Telefon = Text1(3)   'Sätt fältet Telefon i nya posten
.Update               'Spara posten
.Close                'Stäng tabellen
End With

adoConn.Close          'Stäng förbindelser
Set adoRS = Nothing    'Städa upp objekt
Set adoConn = Nothing
```

4.2.2 Använda SQL-sats och metoden Execute() i Connection-objekt

Ett annat sätt att lägga till poster i en tabell är att utföra en uppdateringsfråga (med INSERT-sats). I detta fall använder vi metoden **.Execute()** i Connection-objektet och eftersom det är en uppdateringsfråga så blir det inga resulterande poster.²¹ Vi behöver alltså inget Recordset-objekt och kan använda metoden **.Execute()** som procedur istället för funktion. I samband med att vi exekverar frågan så skickar vi med en variabel (av typen Long) som tilldelas antalet poster som påverkas av frågan.

```
Dim adoConn As ADODB.Connection 'Deklarera variabler
Dim strConn As String, strQuery As String
Dim lngNewRows As Long

'Skapa Connection-objekt, ange anslutningssträng och öppna anslutning
Set adoConn = New ADODB.Connection
strConn = "Provider=MSDASQL;Data Source=ODBCBokning"
adoConn.Open strConn

'Skapa SQL-sats
strQuery = "INSERT INTO tblPersonal VALUES ('bpn', 'Björn', 'Persson', '3678')"
```

4.3 Uppdatera post i tabell

Att uppdatera en post påminner mycket om att skapa en ny post. Här har vi främst tre sätt att göra det på:

- Använda Recordset-objekt.
- Använda SQL-sats och metoden Execute() i Connection-objekt.
- Använda Command-objekt och parametrar.

4.3.1 Använda Recordset-objekt

Skillnaden mellan att lägga till och uppdatera en post med Recordset-objekt ligger i att man först måste hitta existerande posten man vill ändra i. För att hitta posten använder vi en SQL-sats – en SELECT-sats med villkor. I villkoret använder vi (lämpligen) primärnyckeln i tabellen för att garanterat erhålla endast en (eller ingen) post. Innan vi uppdaterar data i posten så måste vi kontrollera att det finns en post, d.v.s. att det fanns en post med motsvarande

²¹ Sanning är att resultatet av funktionen är ett Recordset-objekt som är stängt (samma sak som om metoden **.Close()** hade utförts på objektet). D.v.s. det är onödigt att slösa minne och tid på att deklarera en variabel.

värde i primärnyckeln. För att kontrollera om post finns kan vi t.ex. använda Recordset-objektets egenskap **.RecordCount** (eller egenskaperna **.BOF** och **.EOF**).

```
Dim adoConn As ADODB.Connection
Dim adoRS As ADODB.Recordset
Dim strConn as String, strQuery as String

' skapa Connection-objekt, ange egenskaper och öppna förbindelse
Set adoConn = New ADODB.Connection
strConn = "Provider=MSDASQL.1;Data Source=ODBCBokning"
adoConn.Open strConn

' skapa Recordset-objekt, ange egenskap och kör fråga
Set adoRS = New ADODB.Recordset
strQuery = "SELECT * FROM tblPersonal WHERE Id='bpn'"
adoRS.Open strQuery, adoConn, , adLockOptimistic

If adoRS.RecordCount <> 0 Then 'Om det finns fler poster än 0
  With adoRS
    !FNamn = Text1(1)          'Sätt värden
    !ENamn = Text1(2)
    !Telefon = Text1(3)
    .Update                   'Uppdatera post
  End With
End If

adoRS.Close                  'Stäng
adoConn.Close
Set adoRS = Nothing          'Städa upp
Set adoConn = Nothing
```

4.3.2 Använda SQL-sats och metoden Execute() i Connection-objekt

Givetvis går detta också att lösa genom att köra en uppdateringsfråga (UPDATE-sats) mot datakällan.

```
Dim adoConn As ADODB.Connection
Dim strConn As String, strQuery As String
Dim lngUpdatedRows As Long

strConn = "Provider=MSDASQL.1;Data Source=ODBCBokning"
Set adoConn = New ADODB.Connection
adoConn.Open strConn

strQuery = "UPDATE tblPersonal SET FNamn='Björn', ENamn='Olsson' WHERE Id = 'bpn'"
adoConn.Execute strQuery, lngUpdatedRows

MsgBox "Antal uppdaterade poster: " & lngUpdatedRows

adoConn.Close
Set adoConn = Nothing
```

4.3.3 Använda Command-objekt och parametrar

Koden för Command-objekt och parametrar påminner mycket om koden när vi använder Connection-objektet. Skillnaden ligger i SQL-satsen, bl.a. att vi använder frågetecken istället för värden, samt att vi skapar ett Command-objekt och en parameter för varje frågetecken i SQL-satsen.

Observera att vi **inte** använder några enkla citattecken för att innesluta frågetecknen trots att det är strängar i SQL! Vi behöver inte heller deklarera en variabel för varje parameter utan vi kan återanvända variabeln (adoParam) när vi lagt till den i Command-objektets vektor **.Parameters**.

```
Dim adoConn As ADODB.Connection
Dim adoCmd As ADODB.Command
Dim adoParam As ADODB.Parameter
Dim strConn As String, strQuery As String
Dim lngUpdatedRows As Long

strConn = "Provider=MSDASQL.1;Data Source=ODBCBokning"
Set adoConn = New ADODB.Connection
adoConn.Open strConn

strQuery = "UPDATE tblPersonal SET FNamn=?, ENamn=? WHERE Id = ?"

Set adoCmd = New ADODB.Command
Set adoCmd.ActiveConnection = adoConn
adoCmd.CommandText = strQuery
adoCmd.CommandType = adCmdText      'Ange kommando att utföra
                                     'Ange att SQL-sats

Set adoParam = adoCmd.CreateParameter("FNamn", adWChar, adParamInput, _
                                       Len("bpn"), "bpn")
adoCmd.Parameters.Append adoParam

Set adoParam = adoCmd.CreateParameter("ENamn", adWChar, adParamInput, _
                                       Len("Björn"), "Björn")
adoCmd.Parameters.Append adoParam

Set adoParam = adoCmd.CreateParameter("Id", adWChar, adParamInput, _
                                       Len("Olsson"), "Olsson")
adoCmd.Parameters.Append adoParam

adoCmd.Execute lngUpdatedRows

MsgBox "Antal uppdaterade poster: " & lngUpdatedRows

adoConn.Close
Set adoConn = Nothing
```

5 Felhantering

En applikation utan felhantering är inte en komplett applikation. Det är trevligt om man kan få besked om att ett fel har uppstått och att applikationen inte bara försvinner från skärmen, d.v.s. kraschar. Eller ännu bättre att programmet inte kraschar alls, d.v.s. att programmet återförs till ett stabilt tillstånd efter ett fel.

Generellt sätt brukar man aldrig kunna förutse alla fel och därmed hantera alla situationer. Då är det bra för användarens del, och egen del som programmerare, att kunna informera användaren om vad som gått fel för att kunna hitta en lösning på problemet. Felhantering i Visual Basic 6 är inte en av de bättre, men den är bättre än ingen alls.

Det finns tre huvudsakliga sätt att hantera fel på:

- Att inte hantera fel i metoden.
- Utföra programsats och prova om den utfördes korrekt.
- Använda sig av en felhanterare i metoden.

En fördel med Visual Basics felhantering är att man kan växla mellan olika typer av felhantering under exekveringen av en metod. D.v.s. man kan välja att hantera vissa fel och låta andra fel hanteras av anropande metod (om det finns någon felhantering i den metoden).

Alla typer av felhantering bygger på att man skriver `On Error...` innan koden man vill eller inte vill hantera fel för.

5.1.1 Att inte hantera fel

Detta är den simplaste formen av felhantering – att inte göra något (vilket också är ”standard”). Detta innebär att man kan välja att inte hantera felet i aktuell metod och låta anropande metod hantera felet. Det kan sägas att t.ex. ADO gör på detta sätt. ADO kan t.ex. inte veta att användaren råkade stava fel på primärnyckeln för önskad post och att det därmed inte finns en post att uppdatera.

Om fel uppstår i metod med denna typ av felhantering kommer metoden att avslutas då fel uppstår. Detta medför att ingen form av återställande av ändringar som gjorts i metoden kommer att ske. Applikationen (eller komponenten) kommer eventuellt inte att befinna sig i ett stabilt tillstånd och kan därmed krascha.

Före koden som man inte vill hantera fel för skriver man `On Error Goto 0` (se mer under avsnittet *Att hantera vissa fel men inte andra* nedan).

5.1.2 Att prova om programsats utfördes korrekt

Detta alternativ av felhantering innebär att man använder sig av felhanteringskod enligt exempel nedan. Exemplet bygger på att användaren i textrutan `Text1` anger användareID på person (primärnyckeln i tabellen `tblPersonal`) som användaren vill ändra telefonnummer för, vilket har angivits i textrutan `Text2`. Eftersom frågan i exemplet är felaktig (citattecken saknas) så kommer inte tabellen öppnas och då kan provas om öppning av tabellen har utförts eller inte innan manipulation av poster. I exemplet nedan kommer Else-satsen att utföras.

```
On Error Resume Next      'Om fel, gå till nästa programsats
'Skapa en SQL-fråga där citattecknen (') saknas
strQuery = "SELECT * FROM tblPersonal WHERE Id = " & Text1
adoRS.Open strQuery, adoConn      'Här kommer fel att uppstå!

If adoRS.State = adStateOpen Then  'Om tabellen har öppnats...
    adoRS!Telefon = Text2          '... uppdatera post.
```

```
MsgBox "Posten uppdaterades!"  
Else  
MsgBox "Posten uppdaterades INTE!"      '... annars meddela användare.  
End If
```

Den riktiga SQL-frågan där citattecknen (') finns ska vara:

```
strQuery = "SELECT * FROM tblPersonal WHERE Id = '" & Text1 & "'"
```

Fördelen med denna typ av felhantering är att alla programsatser (som inte genererar fel) kommer att exekveras i metoden. Dock kan ett fel leda till följdfel i metoden. Om en förbindelse till en databas inte kan upprättas så kan inte heller tabell öppnas, vilket leder till att ingen manipulation av data i tabellen kan ske.

Nackdelen med denna typ av felhantering är att det blir många if-satser som kan göra koden svårsläst och metoden kommer att ta längre tid att exekvera. Denna form av felhantering är något av pessimistens felhantering – ”lita inte på att något gick bra”.

5.1.3 Använda sig av felhanterare i metod

Det sista sättet att hantera fel är att skapa en felhanterare för hela metoden. Detta innebär att om det blir ett fel någonstans i metoden så kommer felhanteraren att exekvera. I exemplet nedan har en felhanterare skapats som visar kod för felet samt en beskrivande text (inte alltid den bästa texten dock!). En mer riktig version, än den i exemplet nedan, av felhanteraren kommer innehålla if- eller case-satser. I dessa if-/case-satser hanterar man de fel man kan förutsätta uppstår om t.ex. användare anger felaktiga värden och då kan man ge en mer förklarande beskrivning än den VB/ADO ger, t.ex. på ”vanlig” svenska.

En felhanterare skapas genom att ange en etikett (*label*) som består av en sträng (lämpligen metodens namn plus ”_Error”) och ett kolon (:) efter, t.ex. som i exemplet `btnUpdate_Click_Error`. Innan etiketten bör man skriva `Exit Sub` (eller `Exit Function`) för att avsluta metoden så att inte felhanteraren exekveras om allt gick väl i metoden. För att tala om för VB att man vill hoppa till felhanteraren skriver man enligt texten med grå bakgrund nedan:

```
Private Sub btnUpdate_Click()  
    'Om fel, gå till etiketten btnUpdate_Click_Error:  
    On Error Goto btnUpdate_Click_Error  
    'Skapa en SQL-fråga där citattecknen (') saknas  
    strQuery = "SELECT * FROM tblPersonal WHERE Id = " & Text1  
    adoRS.Open strQuery, adoConn      'Här kommer fel att uppstå!  
  
    'Denna kod kommer aldrig att exekvera  
    'Men vi provar om posten hittades innan vi sätter värde  
    If adoRS.RecordCount <> 0 Then 'Posten har hittats...  
        adoRS!Telefon = Text2      '... uppdatera post.  
        MsgBox "Posten uppdaterades!"  
    Else  
        MsgBox "Posten uppdaterades INTE!" '... meddela användare.  
    End If  
  
    ...  
  
    Exit Sub 'Avsluta metod så att inte felhanterare exekveras  
  
btnUpdate_Click_Error:      'Etikett för felhanterare  
    'Visa dialogruta med nummer för fel och en beskrivning  
    MsgBox "Felkod: " & Err.Number & vbCrLf & Err.Description  
End Sub
```


Om man vill fortsätta exekveringen efter programsatsen som genererade ett fel, och efter att ha hanterat felet, kan man lägga till `Resume Next` sist i felhanteraren. Annars kommer metoden avslutas efter att all kod i felhanteraren har exekverat färdigt.

```
btnUpdate_Click_Error:      'Etikett för felhanterare
    'Visa dialogruta med nummer för fel och en beskrivning
    MsgBox "Felkod: " & Err.Number & vbCrLf & Err.Description
    Resume Next              'Fortsätt med programsatsen efter den
                              'som genererade felet
End Sub
```

5.1.4 Att hantera vissa fel men inte andra

Som nämndes ovan så kan man växla mellan att hantera vissa fel i metod för att sedan strunta i andra fel. I exemplet nedan väljs att hantera fel i början medan den sista koden inte hanteras.

```
Private Sub btnUpdate_Click()
    'Om fel, gå till etiketten btnUpdate_Click_Error:
    On Error Goto btnUpdate_Click_Error
    ...
    On Error Goto 0
    ...
    Exit Sub
    btnUpdate_Click_Error:      'Etikett för felhanterare
    ...
End Sub
```

5.2 Innan felhantering implementera

Innan vi börjar implementera felhantering måste vi först ta reda på vilka tänkbara fel som kan uppstå. Det gör vi lämpligen genom att testa med värden som vi vet är ogiltiga för att ”provocera” fram fel.

D.v.s. om vi börjar implementera felhantering innan vi tagit reda på vilka fel som kan uppstå så blir felsökning svårt att utföra. Lämna därför felhanteringen till sist – men dokumentera gärna (under utveckling) vilka fel som ska hanteras.

5.3 Lite om felsökning

Felsökning (eller avlusning – *debugging*) är en del av programutveckling. Ett (men ganska irriterande) sätt är att använda dialogrutor (`MsgBox()`) för att visa innehåll i t.ex. variabler. Men ett bättre sätt är att använda `Debug.Print()` för att skriva till VB:s Immediate-fönster (längst ner i IDE:n). Som parameter till `Debug.Print()` skickas en sträng. Glöm bara inte att radera innehållet i fönstret med jämna mellanrum så att inte gammal, och eventuellt felaktig, information förvirrar.

(Denna sida har avsiktligt lämnats blank.)

6 Övrigt

6.1 Tekniska detaljer

Universal Data Access (UDA) är en strategi/koncept för **hur** åtkomst av data ska kunna ske. "Implementationen" av UDA levereras (i binärkod) som Microsoft Data Access Components (MDAC). MDAC innehåller teknologier som ODBC, OLE DB och ADO.

Idag (september 2003) är det version 2.7 av MDAC (och dess teknologier i MDAC) som gäller. Version 2.5 (och ev. även 2.1) av MDAC räcker för övningar i denna sammanfattning samt följer med bl.a. Service Pack (SP) 4 för Visual Studio (VS) 6 samt Windows 2000. Om du använder Windows 9x/NT4 utan SP4 för VS6 så kan senaste (och näst senaste) versionen av MDAC hämtas från Microsofts hemsidor på adressen

<http://www.microsoft.com/data/>. För att kontrollera vilken version av MDAC du har kan du högerklicka på filen MSADO15.DLL,²² välja **Egenskaper** och sen klicka på fliken **Version**.

6.2 Komponenter och databaser

Om databaser används i komponenter och många användare kommer att använda komponenten så är det viktigt att tänka på att man öppnar förbindelsen med databasen så sent som möjligt och stänger så fort som möjligt. Antalet förbindelser med databaser är begränsade och för att utnyttja förbindelser så effektivt som möjligt så bör man använda sig av COM/MTS eller COM+ som kan använda sig av en pool av förbindelser. När en förbindelse med en datakälla har använts färdigt av en komponent så placeras dess förbindelse i en pool för andra komponenter att använda sig av förbindelsen.

En, av många, fördelar med att använda COM/MTS eller COM+ är att den kan hantera transaktioner för komponenter utan att programmeraren behöver skriva allt för mycket kod för detta. Transaktioner kan gälla flera tabeller i en eller flera databaser på en eller flera servrar.

6.3 Lagrade frågor

Lagrade frågor (*stored procedures*) i databaser bör i största mån användas för att öka hastigheten på databasaccesser. SQL är **inte** ett effektivt sätt att hämta data från databaser (jämfört lagrade frågor). Lagrade frågor har ofta optimerats i databasen för snabbare behandling.

6.4 Några vanliga fel

6.4.1 Tilldelning av variabler

Ett vanligt fel är att man glömmer använda Set-operatören för att sätt referenser till objekt. T.ex. då man anger Connection-objekt för ett Recordset-objekt:

```
adoRS.ActiveConnection = adoConn      'FEL: adoConn är ett objekt!  
Set adoRS.ActiveConnection = adoConn  'RÄTT: sätt en referens
```

²² Filen heter MSADO15.DLL sen version 1.5 av MDAC trots att den kan innehålla en version 2.x.

6.4.2 Uppdatering av tabell

När man öppnar en tabell eller fråga för att lägga till eller uppdatera en post så måste man komma ihåg att öppna tabellen för att skriva, d.v.s. med rätt typ av låsning:

```
adoRS.Open strQuery, adoConn, adLockOptimistic
```

6.4.3 ODBC-källor

Om man flyttar en applikation eller komponent till en annan dator och har använt sig av en ODBC-källa så får man inte glömma att skapa ODBC-källan (i Kontrollpanelen) på den andra datorn.

6.4.4 ADO-versioner

Även versionen av ADO kan variera på datorer, d.v.s. dator som applikation/komponenter ska exekvera på kan ha en äldre version av ADO. Den senaste versionen av ADO (MDAC) finns att hämta på Microsofts hemsidor.

6.5 Fortsatt läsning

Detta sammanfattning behandlar bara en bråkdel av ADO:s funktioner, och då främst relations-databaser. Ämnen som inte har behandlats är bl.a.:

- ADO och ADSI (*Active Directory Services Interface*), d.v.s. katalogtjänster.
- ADO och CDO (*Collaborate Data Objects*) – att jobba mot t.ex. Microsoft Exchange.
- ADO och XML (*Extensive Markup Language*).

Mer förslag på ämnen, böcker och hemsidor med mer information om databaser (och komponentrelaterade ämnen) finns på min personliga hemsida.