Praxiseinsatz von benutzerdefinierten Klassen in Microsoft® Access

Grundlagen – Beispiele – Tools

Skriptum zum Vortrag auf der AEK7
7. Access-Entwickler-Konferenz
Nürnberg, 25./26.9.2004

Dieses Dokument entspricht im Wesentlichen dem Skriptum, wie es in der AEK-Teilnehmermappe zu finden war. Änderungen betreffen Tippfehler und verbesserungswürdige Formulierungen. Außerdem ist das Inhaltsverzeichnis ausführlicher und das Layout ein wenig luftiger gestaltet.

Feedback jeder Art ist unter unten stehender E-Mail-Adresse ausdrücklich willkommen.

Wien, am 27. September 2004
Paul Rohorzka

softconcept – rohorzka keg
Obere Amtshausgasse 6-8/13
A-1050 Wien
+43 650 softcon (+43 650 7638266)
www.softconcept.at

Inhalt

Wozu Klassen?

Die Klasse, das unbekannte Wesen

Bei aufmerksamer Beobachtung Access-relevanter Seiten im Internet (Community-Seiten, Firmen-Seiten und vor allem Newsgroups und andere Foren) fällt auf, dass Themen der objektorientierten Programmierung recht selten zur Sprache gebracht werden. Auf der einen Seite stehen zwar Entwickler[1], die objektorientierte Konzepte zumindestens im Ansatz wie selbstverständlich in ihren Projekten verwenden, auf der anderen Seite findet sich aber die Vielzahl jener Entwickler, die Klassen höchstens im Rahmen eines Copy- und Paste-Vorgangs in die Finger bekommen[2]. Auf Fragen in Bezug auf derartige Problemstellungen antworten auch meistens nur eine Handvoll Leute.

Was ist nun der Grund, dass gerade im Bereich der Access-Entwicklung objektorientierte Konzepte von vielen Entwicklern aktiv nicht angewandt werden? Vielleicht ist ein Grund darin zu suchen, dass aufgrund der ausgeprägten RAD- (Rapid Application Development) Features von Microsoft® Access der theoretische Unterbau der Entwickler oft nicht in dem Maß ausgeprägt ist, wie bei Anwendern anderer Software-Entwicklungswerkzeuge. Schließlich kann man mit Access selbst mit rudimentären Programmierkenntnissen fürs erste annehmbare Anwendungen erstellen.

Ein weiterer Punkt ist wohl auch die defacto fehlende Hilfestellung der Klassenprogrammierung seitens Access. In der Sammlung der mit Access mitgelieferten Assistenten findet sich kein einziger zur Unterstützung der Klassenprogrammierung. Im VBE (Visual Basic Editor) beschränkt sich die Unterstützung auf das Einfügen von Eigenschafts-Prozeduren (Menü Einfügen - Prozedur). Durch AddIns von Drittanbietern, wie der beliebten Freeware MZ-Tools, dem Class Builder Wizard von Dev Ashish und Terry Kreft oder den Tools von FMS lassen sich zweifelsohne besser geeignete Werkzeuge finden. Deren Funktionsumfang befindet sich dennoch meilenweit hinter dem, was man heutzutage von einem OOD-Tool (Datamodelling und Coding-Tools) erwartet[3].

Vorteile der Verwendung von Klassen

Ist gibt wohl keine Aufgabenstellung in der Anwendungs-Entwicklung unter Access, die sich nicht ohne der Verwendung einer einzigen Klasse realisieren ließe. Für viele Probleme existieren Lösungen, die oft auf Sammlungen von Funktionen und Prozeduren in einem oder mehreren Standard-Moduln basieren.

Was sind aber nun die Vorteile, die der Entwickler aus der Entwicklung mit benutzerdefinierten Klassen ziehen kann? Ich möchte einige hier anführen:

- Klassen können ohne weiteres Zutun des Entwicklers Standard-Verhalten bieten. Bei Funktionen könnte dieser Zustand nur durch viele optionale Parameter erreicht werden. Das ist sowohl für den Entwickler der Funktionen als auch für den Verwender der Funktionen mit einigem Aufwand verbunden.
- Die Verwendung von Klassen-Eigenschaften statt Funktionsparametern liefert automatisch besser lesbaren Quellcode.

  Beispiel einer prozeduralen Implementierung:

```
Call NeuerArtikel("Seife", "Toiletteartikel", 0.79, 100, 30, 2)
```

[1] Der Begriff "Entwickler" soll in diesem Skriptum auch unausgesprochen die zum Glück vertretenen weiblichen Entwicklerinnen mit einschließen.

[2] "Nimm dazu doch die Klasse clsXxx von Xxx. Du kannst sie dir unter http://www.Xxx.Xxx/Xxx herunterladen!"

[3] Hier sei stellvertretend IBM Rational („Rational Rose") genannt, auch wenn das im vorliegenden Zusammenhang beinahe schon ein hinterhältiger und unzulässiger Vergleich ist. Schließlich spielen solche Tools in einer völlig anderen Liga.

Der selbe Vorgang unter Zuhilfenahme eines Objekts könnte so aussehen:

```
With artNeu
   .Bezeichnung = "Seife"
   .Warengruppe = "Toiletteartikel"
   .Preis = 0.79
   .Lagerstand = 100
   .Mindestlagerstand = 30
   .Lager = 2
End With
```

- Bei Aufgabenstellungen, in den Zwischenergebnisse vorgehalten werden müssen, wird es eventuell sehr mühsam, da ein Mechanismus bereitgestellt werden muss, der die jeweils richtigen Zwischenergebnisse an die Funktionen übergeben werden. Anwendungen, die dies verdeutlicht, sind die Klasse `CMoveAndSizeLabel` (siehe Seite 40) oder die Klasse `CFormPosSize` (siehe Seite 41).
- Features, bei denen auf Ereignisse von Steuerelementen oder Formularen reagiert werden muss, lassen sich nur mit Hilfe von Klassen sinnvoll realisieren. Andere Möglichkeiten, wie der direkte Aufruf von Funktionen in den Ereignis-Eigenschaften sind in Hinblick auf Fehlersuche und Dokumentation für die Anwendung im großen Stil kaum praktikabel.
- Access ist von Haus aus ein objektbasiertes System, das dem Entwickler viele eingebaute Objekte zur Verfügung stellt (`Form`, `Report`, `TextBox`, `CommandButton`, `DoCmd`, …). Entwickler sind gewohnt, mit diesen Objekten zu arbeiten. Warum soll es dann nicht ebenso möglich sein, objektbasiert auf die eigentlichen Nutzdaten der Anwendung zuzugreifen? Diese Möglichkeit besteht unter Berücksichtigung der Einschränkungen durch VB(A) und Access.

Über dieses Skriptum

Ich habe für den vorliegenden Text einen bewusst pragmatischen Zugang gewählt. Es geht mir nicht um das vollständige Abdecken Objektorientierter Konzepte in allen Varianten und Ausprägungen. Vielmehr möchte ich die durchwegs positiven Erfahrungen aus meiner Arbeit als Entwickler unter Access in Bezug auf benutzerdefinierte Klassen dem Leser näherbringen.

Ich weiß, dass es mir nicht gelingen wird, alle Skeptiker des Themas (und davon soll es ja einige geben) zu überzeugen und möchte die Verwendung von Klassen auch nicht als Allheilmittel für alle Access-Probleme verkaufen. Ich möchte hingegen einen Beitrag leisten, die Arbeit manches Access-Entwicklers vielleicht ein wenig effizienter und vielleicht sogar einfacher zu gestalten.

Der Vortrag auf der AEK7, dessen Skriptum Sie in Händen halten, soll somit ein kleiner Impuls sein, die deutschsprachige Community verstärkt auf die Vorteile der Entwicklung auf Basis objektorientierter Konzepte auch unter Access aufmerksam zu machen.

Szenarien bei der Entwicklung mit Access

Anforderungen

Implementierungen

Probleme

Visionen

Einleitung

In diesem Abschnitt möchte ich einige typische Szenarien bei der Entwicklung mit Microsoft® Access anführen und die Probleme, Lösungsmöglichkeiten und Visionen in diesem Zusammenhang beleuchten.

Problembereich 1: Konsistente GUI

Szenario

Der Großteil der Formulare in einer Anwendung hat viele Eigenschaften und Anforderungen gemeinsam:

- **Konsistentes GUI** (Farben, Beschriftungen, Positionen, ...)
- Durch den Endbenutzer in gewissem Rahmen **konfigurierbares GUI**
- **Wiederherstellen** der Formulare in der zuletzt eingestellten **Größe**, an der letzten **Position**
- Sinnvolle Ausnützung des Bildschirms durch **größenänderbare Formulare** (*nicht* die Anpassung an verschiedene Auflösungen mit Skalierung des gesamten Inhalts, sondern **größere Arbeitsbereiche**, z.B. Listen bei größerem Fenster, **Verankern** von Steuerelementen am **rechten/unteren Rand**)

Schwierigkeiten

All diese Features lassen sich implementieren, aber:

- Der **Code** dazu muss in jedem Formular sehr ähnlich **wiederholt** werden (Copy, Paste & Modify)
- Gewünschte **Änderungen** des Verhaltens müssen **im Code jedes Formulars** durchgeführt werden, mit allen Fehlern, die dabei passieren, Inkonsistenzen, die sich damit einschleichen.
- **Neues Verhalten** muss **in jedem Formular nachgerüstet** werden.

Visionen

- **Implementieren** des Codes **an zentraler Stelle**
- Einfache **Anwendung** dieses Codes auf alle **erforderlichen** Formulare
- Anpassungen und Erweiterungen an zentraler Stelle

Problembereich 2: Standard-Reaktionen auf Ereignisse

Szenario

Die meisten Formulare einer Anwendung haben einige Features gemeinsam:

- **Standard-Schaltflächen** (Ok, Abbrechen, Übernehmen, Neu)
- **Navigation** (Auswahlliste, Such-Kombinationsfeld, **History**, Vor- Zurück, ...)
- Behandlung fehlender oder **fehlerhafter Eingaben**
- Anzeigen des **zuletzt** angezeigten **Datensatzes**

Schwierigkeiten

Auch hier ist die Implementierung zweifellos möglich, jedoch mitunter sehr mühsam und vor allem fehlerträchtig. Einige Schwierigkeiten im vorliegenden Problembereich:

- Anpassung an Bezeichnungen der zugrundeliegenden Tabelle (Tabellenname, Primärschlüsselfeld, Felder mit Klartext zum Suchen, etc.)
- Änderungen bei den erforderlichen Feldern erfordern beim Wunsch nach aussagekräftigen Meldungen ziemliche Arbeit.

Visionen

- Erweitertes Standardverhalten von Formularen an zentraler Stelle wart- und änderbar
- Anpassung an Änderungen in der Datenstruktur an möglichst wenig stellen im Code

Problembereich 3: Auswertung/Manipulation von Daten per Code

Szenario

Zugriff auf die **Daten** der Datenbank. Einfachster Fall unter Verwendung einer fragwürdigen Domänen-Aggregatfunktion:

```
strLastName = DLookup("Cust_LastName", _
                      "tblCustomers", _
                       "CustomerID = " & lngCustomerID)
```

Alternative: Verwendung eines Recordsets (auch noch ohne Fehlerbehandlung!):

```
Dim rstCustomers As DAO.Recordset

Set rstCustomers = CurrentDb().OpenRecordset( _
   "SELECT * FROM tblCustomers WHERE CustomerID = " & _
                 lngCustomerID)

If Not rstCustomers.EOF Then
   strLastName = Nz(rst.Fields("Cust_LastName").Value,"")
End If

If Not (rstCustomers Is Nothing) Then rstCustomers.Close
Set rstCustomers = Nothing
```

Schwierigkeiten

- Wird oft und an verschiedensten Stellen in der Anwendung benötigt
- **Mühsam** und **fehleranfällig**
- **Änderungen an der Datenstruktur** haben oft schwer absehbare **Konsequenzen für den Code** in den einzelnen Formularen.
- Die **Einhaltung von Geschäftsregeln** muss jedesmal neu sichergestellt sein.
- **Änderungen in den Geschäftsregeln** müssen **an allen relevanten Stellen** (wo sind *alle*?) vorgenommen werden.

Visionen

Ideal wäre ein tatsächlich **objektbasierter Zugriff** auf die Daten:

```
strLastName = Customers(lngCustomerID).LastName
```

Vorteile:

- **Unabhängig von Bezeichnungen der Tabellen und Felder**, sogar unabhängig von der tatsächlichen Herkunft (in einem Feld einer Tabelle oder durch Berechnungen aus anderen Feldern)
- Code ist weit unabhängiger von Details der Datenspeicherung (Struktur, Datenbank, ...)
- Die Objekte können die **Einhaltung von Geschäftsregeln erzwingen**

Ausführlicheres Beispiel:

```
With Customers(lngCustomerID)
  MsgBox "Der Kunde " & .PrettyName & _
         " hat die Rechnung Nr." & .Invoices(InvID).No & _
         " bereits am " & .Invoices(InvID).Paid & _
         " bezahlt." & vbNewLine & _
         "Die " & .Invoices(InvID).Details.Count & _
         " Posten wurden am " & _
         .Invoices(InvID).Delivery.ShipDate & _
         " geliefert."
End With
```

Geht das?

Lassen sich diese Anforderungen in die Praxis umsetzen?
Mit welchem Aufwand ist eine Umsetzung verbunden?
Kann ich das nicht auch alles ohne Klassen erreichen?

Einführung in das Programmieren mit Klassen in VB(A)

Grundlagen

Elemente

Schnittstellen

Objekte in VB(A)

Objekte in Access

Grundlagen

Im Allgemeinen lässt sich die Funktion jeder Software auf folgende abstrakte Weise beschreiben: Zu verarbeitende Daten werden durch Steuerung (meist eines Benutzers) in Ergebnis-Daten transformiert. Beispielsweise wird aus den Daten einer Datenstruktur mit Kunden-, Artikel- und Bestelldaten eine Rechnung generiert. Aus der Sicht des Software-Entwicklers stellt sich diese Aufgabenstellung als Zusammenspiel einer Vielzahl von einzelnen Verarbeitungsschritten dar. Jeder dieser Schritte ist für die Erledigung eines im Idealfall klar abgegrenzten Teils des Gesamtproblems zuständig. Für diese Teil-Aufgabenstellung ist ein Teil der Daten notwendig, und das Ergebnis wird mittelbar oder unmittelbar zum Ergebnis des gesamten Prozesses beitragen. Die effektive Aufgabenteilung und Zusammenführung der Einzelergebnisse ist wiederum Teil der Aufgabe übergeordneter Prozeduren.

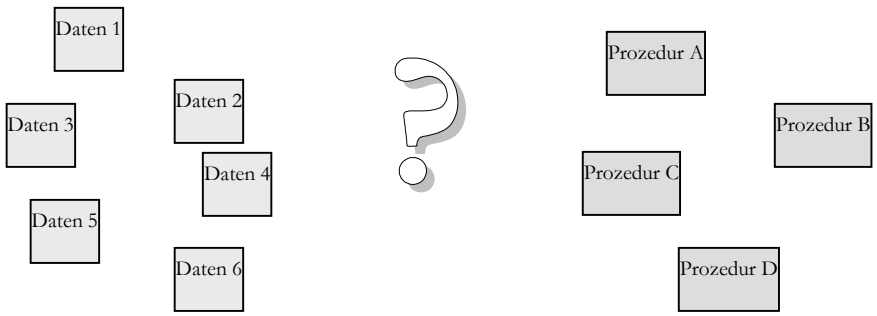
Wie die konzeptionelle Umsetzung dieser Struktur aussehen kann, soll nun im Folgenden näher betrachtet werden.

Prozedurale Programmierung

Bei der klassischen Prozeduralen Programmierung, wie sie von Sprachen wie Pascal, C, Basic, etc. unterstützt wird, liegt es in der Verantwortung des Entwicklers, im Sinne obiger Überlegung die richtigen Funktionen mit den richtigen Daten zu beschicken sowie die Ergebnisse der Teil-Funktionen korrekt zusammenzuführen:

Für die Entwicklung mit VB(A) bedeutet dies, dass der Entwickler zur Erfüllung einer Teilaufgabe die Variablen mit den zu verarbeitenden Daten in der vorgesehenen Art und Weise an die passende Funktion übergeben muss, und die von ihr erhaltenen Daten wieder entsprechend weiter verarbeiten muss.

Dieser Vorgang ist wenig intuitiv. Die alltägliche Situation "Fahrrad fahren" würde nach prozeduraler Vorgangsweise als Anwenden der Aktion "Fahren" auf die Daten "Fahrrad" formuliert werden können (Pseudo-Syntax: `Fahren(Fahrrad)`). Eine unzulässige Zuordnung wäre aber die Aktion "Fahren" auf die Daten "Papierstoß" anzuwenden (Pseudo-Syntax: `Fahren(Papierstoß)`).



Fehler wie dieser, nämlich die Übergabe unpassender Daten an eine Funktion lassen sich im Allgemeinen nicht bereits zur Übersetzungszeit (vom Compiler) finden, sondern treten entweder erst zur Laufzeit auf oder zeigen sich lediglich in falschen Ergebnissen. Vom Compiler können nur jene Fehler gefunden werden, die inkompatible Datentypen beteiligt sind. Gerade in diesem Zusammenhang sind die Fähigkeiten der impliziten Typumwandlung des VB(A)-Compilers eher ein Problem als ein Segen.

Da gerade in Datenbank-Anwendungen sehr oft mit Primärschlüsselwerten in Form von AutoWerten vom Datentyp `Long` gearbeitet wird, ist das Problem der unpassenden Daten besonders dramatisch. Nehmen wir an, es gibt eine Prozedur `Bestellen`, die als Parameter den zu bestellenden Artikel in Form der `ArtikelID`, die gewünschte Anzahl in Form eines `Integer`-Parameters und den Lieferanten in Form der `LieferantenID` erwartet.

Diese wäre eine korrekte Verwendung dieser Prozedur:

```
Call Bestellen(lngArtikelID, intStueck, lngLieferantenID)
```

`lngArtikelID`, `intStueck` und `lngLieferantenID` seien `Long`- bzw. `Integer`-Variablen mit entsprechenden Werten.

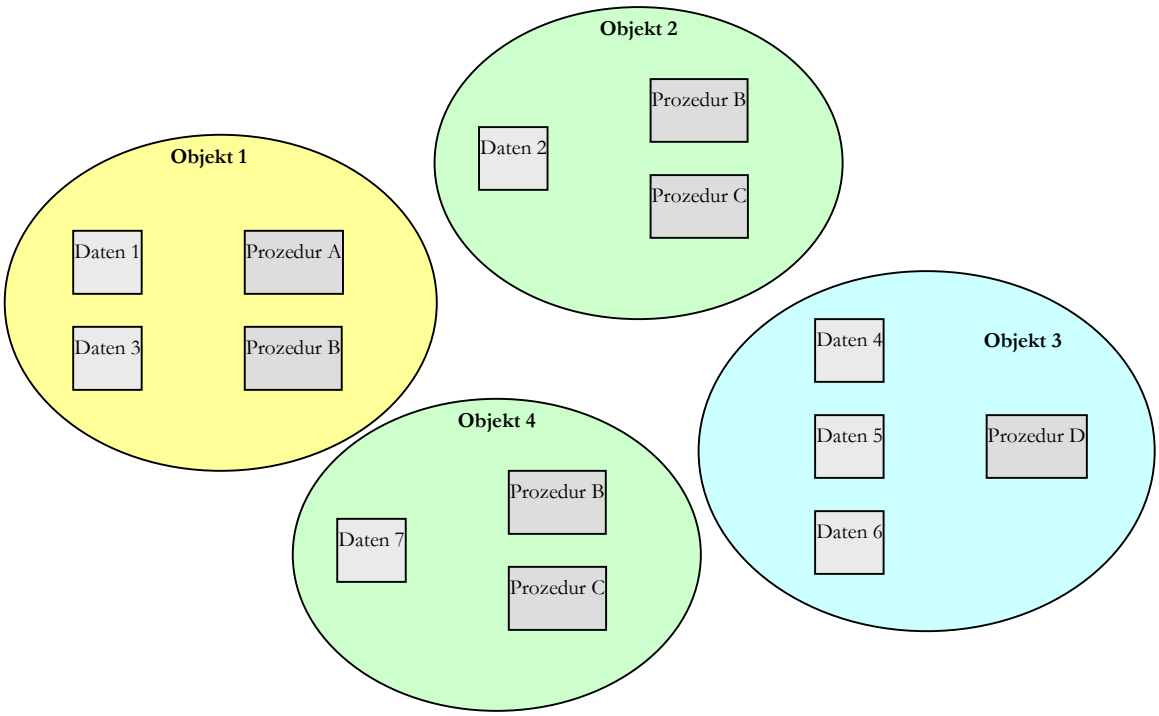
Es ist für den Compiler nun völlig korrekt, ruft der Entwickler diese Funktion mit völlig unpassenden Daten auf:

```
Call Bestellen(lngKundenID, Day(Now()), lngDruckerID)    ' sinnlos!
```

Diese semantisch absolut sinnlose Verwendung der Prozedur `Bestellen` zeigt erst zur Laufzeit Probleme. Diese Probleme zeigen sich bestenfalls in Form einer Verletzung der referenziellen Integrität, die in einen Laufzeitfehler mündet, schlimmstenfalls aber lediglich in fehlerhaften Daten.

Objektorientierter/-basierter Ansatz

Der objektorientierte Ansatz umgeht nun das Problem der Anwendung der korrekten Prozeduren auf die passenden Daten durch die Zusammenfassung der Daten mit den zugehörigen Funktionen. Eine solche Kombination von konkreten Daten mit den dazugehörenden Funktionen heißt **Objekt**:



Klassen

Wie ein konkretes Objekt aufgebaut ist, also aus welchen Elementen (im Wesentlichen Daten und Prozeduren) es besteht und wie diese Elemente miteinander interagieren (also der Programmcode), wird durch seine **Klasse** bestimmt. Eine Klasse ist also sozusagen die Vorlage für die Erstellung eines konkreten Objekts. Man spricht auch davon, dass ein konkretes Objekt eine **Instanz** der zugrundeliegenden Klasse ist. Der Vorgang der Erstellung eines konkreten Objekts heißt daher auch **instanziieren**.

Eine Klasse verhält sich zu den nach ihr instanziierten Objekten ähnlich einem Datentyp zu einer konkreten Variable dieses Datentyps.

Diese Kombination aus Daten und Prozeduren kommt dem menschlichen besser Denken entgegen. Angewandt auf das obige Beispiel "Fahrrad fahren" würde das bedeuten, auf das Objekt "Fahrrad" sehr wohl die Aktion "Fahren" anwenden zu können (VB-Syntax: `Fahrrad.Fahren`), jedoch nicht auf das Objekt "Papierstoß", da dieses Objekt keine Aktion "Fahren" bietet (stattdessen vielleicht eine Aktion "Sortieren" – VB-Syntax: `Papierstoß.Sortieren`). Fehlerhafte Kombinationen können nun bereits vom Compiler erkannt werden, da im Gegensatz zu `Fahrrad.Fahren` der Ausdruck `Papierstoß.Fahren` nicht korrekt sein kann – die Klasse des Objekts `Papierstoß` bietet keine Methode namens `Fahren`.

Mehrere Instanzen der selben Klasse

In obiger Grafik sehen wir, dass von der selben Klasse selbstverständlich mehrere Instanzen existieren können. Objekt 2 und Objekt 4 stellen offenbar Instanzen der selben Klasse dar. Gemeinsam sind diesen beiden Objekten daher die Prozeduren (Prozedur B und Prozedur C). Unterschiedlich sind jedoch die Daten (Daten 2 und Daten 7). Es ist zu beachten, dass bei der tatsächlichen Implementierung des Klassenkonzepts der Code der Prozeduren und Funktionen (die sogenannten "Methoden" – siehe unten) einer Klasse nur ein einziges Mal im Speicher existiert, gleich wieviele Instanzen dieser Klasse angelegt werden. Um die Methoden mit den korrekten Daten arbeiten zu lassen, wird jeder Methode einer Klasse beim Aufruf ein zusätzlicher versteckter Parameter übergeben, der einen Verweis auf die Daten der zugehörigen Instanz der Klasse enthält.

Beim objektorientierten Ansatz wird die komplette in einem Objekt steckende Komplexität vor dem Anwender (i.e. dem Entwickler, der mit dieser Klasse programmiert) verborgen. Der Anwender muss sich also im Allgemeinen nicht darum kümmern, woher das Objekt seine Daten erhält, bzw. was es mit seinem Zustand tut. Es ist für den Anwender auch unerheblich, wie in der Klasse eine bestimmte Funktionalität implementiert ist ("Blackbox-Prinzip", Datenkapselung).

Elemente

Jede Klasse kann aus folgenden Elementen bestehen: Eigenschaften, Methoden, Ereignissen

Eigenschaften

Eigenschaften repräsentieren den Zustand (die Daten) eines Objekts.

Beispiele:

- Das Gewicht eines Fahrzeugs
- Die Farbe eines Steuerelements

Methoden

Unter dem Begriff Methoden werden die **Funktionen** und **Prozeduren** des Objekts zusammengefasst.

Beispiele:

- Reparieren eines Fahrzeugs
- Drucken eines Berichts

Ereignisse

Unter COM (Component Object Model) und damit VB(A) können Objekte andere Objekte vom **Eintritt bestimmter Zustände** informieren. Diese Information kann mittels Ereignissen übermittelt werden.

Beispiele:

- Das Licht des Fahrzeugs ist ausgefallen
- Das Steuerelement hat den Fokus verloren

Schnittstellen (Interfaces)

Schnittstellen (Interfaces) beschreiben eine **Schnittstelle** einer Klasse nach außen. Über die Elemente der Schnittstelle können übergeordnete Programmteile mit dem Objekt in Kontakt treten.

Eine Schnittstelle ist definiert durch eine Gruppe von

- öffentlichen Eigenschaften
- öffentlichen Methoden

Durch Schnittstellen können Objekte unterschiedlicher Klassen auf eine einheitliche Art und Weise angesprochen und verwendet werden, sofern sie die durch die Schnittstelle definierten Standards erfüllen. Man spricht hier davon, dass die **Klasse** eine **Schnittstelle implementiert** ("Schnittstellenvererbung").
Eine Klasse kann mehrere Schnittstellen implementieren. Beispielsweise kann eine Klasse `CPerson` sowohl ein Interface `IMitarbeiter` als auch ein Interface `IKunde` implementieren. Somit kann ein Objekt der Klasse `CPerson` einmal als Mitarbeiter und einmal als Kunde betrachtet werden. Jede Klasse implementiert automatisch ein Standardinterface, das aus allen öffentlichen Eigenschaften und Methoden der Klasse besteht.

Der große Vorteil der Schnittstellen wird offenbar, wenn versucht wird, **Programmteile** nicht für Objekte bestimmter Klassen, sondern **für Schnittstellen** zu schreiben. Verlangt beispielsweise ein Programmteil zur Lohnverrechnung von den zu bearbeitenden Objekten lediglich, dass sie Schnittstelle `IMitarbeiter` implementieren, ist es unerheblich, von welchen konkreten Klassen die übergebenen Objekte instanziiert wurden. Es könnten zum Beispiel sowohl die Klasse `CAngestellter` als auch die Klasse `CArbeiter` die Schnittstelle `IMitarbeiter` implementieren. Für den Programmteil zur Lohnverrechnung ist es nun gleichgültig, ob ein übergebenes Objekt eine Instanz der Klasse `CAngestellter` oder der Klasse `CArbeiter` ist. Der Programmteil betracht beide Objekttypen durch die Elemente der gemeinsamen Schnittstelle `IMitarbeiter` ("Polymorphismus").

Durch das **Entwickeln gegen Schnittstellen** statt gegen konkrete Klassen kann folgender Nutzen gezogen werden:

- Der **Code** funktioniert **für viele** unterschiedliche **Implementierungen**
- Relevante Mengen von Code können über weite Strecken **Copy&Paste-fähig** gemacht werden. Klassen-Spezifika finden sich oft nur mehr im Deklarationsabschnitt, aber nicht im eigentlichen Code.

Objekte in VB(A)

Wert- vs. Referenztypen

Werttypen

Wird eine Variable eines eingebauten VB-Datentyps, Aufzählungstypen oder Benutzerdefinierten Datentyps (UDT) erstellt, so repräsentiert der Variablenbezeichner in allen weiteren Zugriffen den aktuellen Wert dieser Variablen:

```
Dim curUmsatz As Currency
Dim curGewinn As Currency
Dim curKosten As Currency

' ...

curUmsatz = 123.5   ' Der Wert der Variablen curUmsatz
                    ' wird auf 123,50 gesetzt
curGewinn = curUmsatz – curKosten ' curUmsatz steht für 123,50
```

Solche Variablen werden daher auch als Werttypen bezeichnet. Zwei Variablen können nicht den selben Speicherplatz benennen.
Zuweisungen an Variablen von Werttypen erfolgen im Allgemeinen durch Zuweisung mittels des Operators =:

```
intAnzArbeitstage = intAnzGesamtTage - intAnzFreieTage
```

Referenztypen

Bietet eine Variable Zugriff auf eine Instanz einer Klasse (also auf ein Objekt), so sprechen wir von einer **Objektvariablen**. Sie stellt lediglich eine Referenz (einen Verweis) auf das eigentliche Objekt dar.
Die Reservierung des Speicherplatzes für den Objektverweis erfolgt mittels Dim-Anweisung:

```
Dim objChef As CMitarbeiter
```

Es sind natürlich auch die Varianten mit Public bzw. Private (auch Global und Friend) möglich.
Eine Erweiterung dieser Syntax ist für Objekte, die Ereignisse auslösen können möglich. Siehe dazu Abschnitt "Ereignisse" auf Seite 19.
Beachten Sie, dass diese Anweisung keinen Speicherplatz für das Objekt selbst reserviert. Dies kann zwar mit einer Erweiterung der Dim-Anweisung um das Schlüsselwort New erreicht werden, was aber gravierende Nachteile mit sich bringt. Näheres dazu siehe Anhang ("Don't: Dim ... As New …" auf Seite 49).

Zuweisungen an Objektvariablen erfolgen über die Set-Anweisung:

```
Set objChef = Employees(lngChefEmpID)
```

Mehrere Objektvariablen können auf das selbe Objekt verweisen:

```
Set objMitarbeiterDesMonats = objJudith
Set objChef = objJudith
```

In obigem Beispiel halten beide Variablen eine Referenz auf ein und dasselbe Objekt, nicht auf getrennte Kopien zweier Instanzen.

Explizites Beenden einer Referenzierung

Referenzierungen werden durch Zuweisung des Schlüsselwortes Nothing aufgehoben:

```
Set objChef = Nothing
```

Wird die Objektvariable nicht explizit auf Nothing gesetzt, so wird der Verweis aufgehoben, wenn die Objektvariable ihren Gültigkeitsbereich verlässt. Darauf sollte man sich aber besser nicht verlassen (siehe folgender Abschnitt).

> Beachten Sie in diesem Zusammenhang bitte auch unbedingt den Access-Bug in Bezug auf WithEvents ("Ein Access-Bug" auf Seite 39).

Vergleich zweier Objektverweise

Zum Vergleich zweier Objektverweise dient der Operator `Is`:

```
If objKunde1 Is objKunde2 Then
   Debug.Print "Kunde 1 ist Kunde 2!"
Else
   Debug.Print "Kunde 1 ist ungleich Kunde 2!"
End If
```

Bestimmen, ob Verweis gesetzt ist

Manchmal ist es notwendig herauszufinden, ob eine Objektvariable eine gültige Referenz enthält, oder noch keiner Instanz zugewiesen wurde. Dies kann mittels eines Vergleichs der Objektvariable mit dem speziellen Wert `Nothing` erreicht werden:

```
If objChef Is Nothing Then
   ' ...
End If
```

Die Lebensdauer eines Objekts

Die Lebensdauer eines Objekts wird nicht wie bei Werttypen durch den Gültigkeitsbereich der Objektvariablen bestimmt. Da Visual Basic eine auf COM basierende Sprache ist (manche sagen, es sei *die* COM-Sprache schlechthin), wird für das Erstellen und Zerstören von Objekten der Mechanismus von COM verwendet. Ein

grundlegendes Verständnis dieses Mechanismus ist daher für die saubere Programmierung mit Objekten von Nöten.

Erstellen eines Objekts

Eine Instanz einer Klasse (ein Objekt) wird mit dem Schlüsselwort `New` erstellt:

```
Set frmKundenA = New Form_frmKunden
```

Mit dieser Anweisung werden zwei Dinge initiiert. Zum einen wird mit Hilfe des Schlüsselwortes `New` eine neue Instanz der Klasse `Form_frmKunden` im Speicher angelegt. Zum anderen verweist nun die Objektvariable `frmKundenA` auf diese neu erstellte Instanz von `Form_frmKunden`.

Zu diesem Zeitpunkt wird auch das Ereignis `Initialize` des Klassenmoduls ausgelöst. Näheres dazu siehe Abschnitt "Konstruktion und Destruktion" auf Seite 29.

Referenzzähler

Jedes COM-Objekt (und damit auch jedes VB(A)-Objekt) hält einen sogenannten Referenzzähler. Dieser führt darüber Buch, wieviele Objektvariablen einen Verweis auf das Objekt halten. Bei jedem Setzen eines Verweises auf das Objekt wird der Zähler um 1 erhöht, bei jedem Löschen eines Verweises auf das Objekt wird er um 1 erniedrigt. Nach einer Zeile `Set ... = New ...` ist der Referenzzähler des Objekts mit dem Wert 1 initialisiert.

Löschen von Objekten

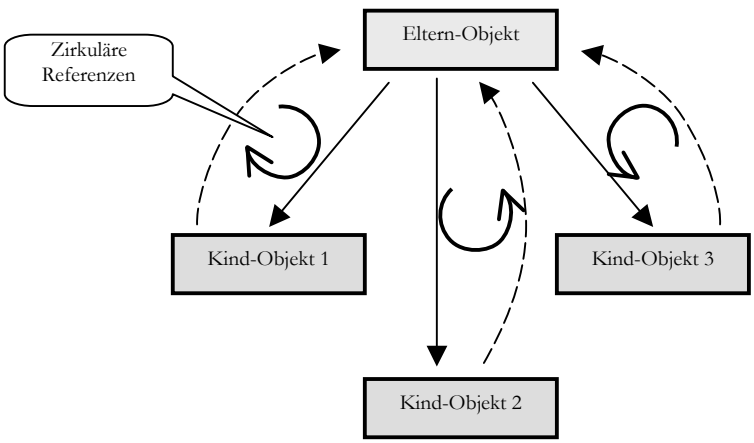
Objekte können in COM nicht direkt gelöscht werden. Wird jedoch der letzte Verweis auf das Objekt gelöscht, erreicht der Referenzzähler den Wert 0 und das Objekt wird automatisch aus dem Speicher entfernt. Verweise werden entweder explizit durch Setzen des Verweises auf Nothing oder implizit bei Verlassen des Gültigkeitsbereichs der Objektvariablen gelöscht.

Zu diesem Zeitpunkt wird auch das Ereignis Terminate des Klassenmoduls ausgelöst. Näheres dazu siehe Abschnitt "Konstruktion und Destruktion" auf Seite 29.

> Hinweis: Objektverweise sollten immer paarweise gesetzt und wieder gelöscht werden.
> Ausführliche Informationen zu diesem Theme finden Sie im Anhang ("Paarweises Setzen und Löschen von Verweisen" auf Seite 48).

Zirkuläre Referenzen

Oft ist es praktisch in einer Objekthierarchie bei untergeordneten Objekten eine **Parent-Eigenschaft** als Verweis auf das übergeordnete Element zu haben. Da jedoch fast immer auch das übergeordnete Element einen Verweis auf seine Kinder hält, entsteht eine zirkuläre Referenz:



Das Problem dabei ist nun, dass das Eltern- und seine Kindobjekte einander wechselseitig referenzieren und auf diese Art einander am Freigeben der Referenzen und damit am Zerstören der Objekte hindern.

Abhilfe

Um dennoch eine korrekte Freigabe der reservierten Speicherbereiche zu erhalten, kann eine Methode zum Aufbrechen der zirkulären Referenz verwendet werden:

In der Kind-Klasse:

```
Public Sub TearDown()
   Set mp_objParent = Nothing
End Sub
```

In der Eltern-Klasse:

```
Public Sub TearDown()
   Dim objChild As CCHild

   For Each objChild In m_collChildren
      objChild.TearDown
   Next objChild
End Sub
```

Aufruf von TearDown vor dem Setzen der Objekt-Referenz auf Nothing:

```
objTop.TearDown        ' Alle Parent-Verweise der Kinder löschen
Set objTop = Nothing   ' Verweise auf Kinder werden gelöscht
```

Nachteilig bei dieser Lösung bleibt, dass es im Verantwortungsbereich des Entwicklers liegt, die Methode zum Aufbrechen der zirkulären Referenz (hier: `TearDown`) auch tatsächlich aufzurufen.

Syntax für den Zugriff auf Elemente

Im Folgenden wird die Syntax des Zugriffs auf die unterschiedlichen Element-Typen eines Objekts besprochen.

Eigenschaften

Eigenschaften eines Objekts werden in VB mit Hilfe des Punkt-Operators angesprochen[4]:

```
<Objekt>.<Eigenschaft>
```

Der Zugriff auf eine Eigenschaft kann im Allgemeinen schreibend und lesend erfolgen. Bei der Erstellung benutzerdefinierter Klassen kann der Zugriff aber auch nur schreibend oder nur lesend gestaltet werden.

Beispiele:
Schreibzugriff auf die Eigenschaft `Caption` des Objekts `lblInfo`:

```
lblInfo.Caption = "Keine weiteren Datensätze vorhanden."
```

Lesezugriff auf die Eigenschaft `Column` des Objekts `cmdKunden`:

```
strMsg = "Kunde " & cmbKunden.Column(2) & " wird nun gelöscht."
```

Die Eigenschaft `Column` des Listbox- (und damit auch des Kombinationsfeld-) Steuerelements in Access ist übrigens der eher seltene Fall einer parametrisierten Eigenschaft, die darüber hinaus auch noch einen optionalen Parameter verwendet. Genaueres dazu siehe Abschnitt "Parametrisierte Eigenschaften" auf Seite 27.

[4] Erklärungen zur Syntax-Beschreibung siehe Anhang "Syntax-Auszeichnungen" auf Seite 48.

Methoden

Methoden eines Objekts werden ebenfalls über den Punkt-Operator angesprochen:

```
<Objekt>.<Methode>([<Parameterliste>])
```

Beispiele:
Aufruf der parameterlosen Methode `SetFocus` des Objekts `txtNachname`:

```
txtNachname.SetFocus
```

Aufruf (mit Parameterübergabe) der Methode `OpenForm` des Objekts `DoCmd`:

```
DoCmd.OpenForm "frmKunden"
```

Ereignisse

Wenn auf die von einem Objekt ausgelösten Ereignisse reagiert werden soll, muss die Deklaration der Objektvariablen um das Schlüsselwort `WithEvents` ergänzt werden:

```
Dim WithEvents <Objekt> As <Klasse>
```

Tritt nun ein Ereignis ein, wird – sofern vorhanden – eine wie folgt aufgebaute Ereignis-Prozedur aufgerufen:

```
Sub <Objekt>_<Ereignis>([<Parameterliste>])
```

Beispiele:
Ereignisprozedur für das Ereignis `Click` des Objekts `cmdClose`:

```
Private Sub cmdClose_Click()
   DoCmd.Close acForm, Me.Name
End Sub
```

Abfangen des Ereignisses `Close` einer Instanz von `Word.Document`:

```
Private WithEvents m_docWord As Word.Document
' ...
Private Sub m_docWord_Close()
   DbLogger.Log m_docWord.FullName
End Sub
```

> Beachten Sie hier bitte unbedingt den Access-Bug in Bezug auf WithEvents (“Ein Access-Bug” auf Seite 39).

Standard-Element

Jede Klasse kann eine Eigenschaft oder Methode als Standard-Element kennzeichnen. In diesem Fall kann beim Zugriff die Angabe des Namens des Standard-Elements entfallen.

Beispiel:
Zugriff auf die Standard-Eigenschaft "Caption" eines Label-Steuerelements "lblInfo":

```
' (Optionale) Angabe des Standard-Elements
lblInfo.Caption = "Es sind noch drei Plätze frei."
' Ohne Angabe des Standard-Elements
lblInfo = "Es sind noch drei Plätze frei."
```

Die Unterscheidung, ob das Objekt selbst, oder sein Standard-Element gemeint ist, kann nur anhand der Art der Zuweisung erfolgen. Bei einer Zuweisung mit Set ist das Objekt, ohne Set das Standard-Element gemeint:

```
Dim txtCurrent As Access.TextBox
Dim strValue As String

Set txtCurrent = Me.txtNachname  ' Das Objekt selbst
strValue = Me.txtNachname        ' Das Standard-Element (Value)
```

Das Standard-Element kann auch eine Methode sein. In diesem Fall ruft die Angabe des Namens des Objekts allein diese Standard-Funktion bzw. -Prozedur auf.

Access-Objekte

Bei der Entwicklung mit Hilfe von Ereignisprozeduren in Access werden an vielen Stellen Objekte verwendet. Dies fällt manchmal gar nicht sonderlich auf, weil die Objekte meistens ein Oberflächen-Element kapseln. Insbesondere die Instanziierung dieser Objekte bleibt hier dem Entwickler verborgen. Die Objekte sind einfach da, weshalb man sie meist kaum von den zugehörigen Oberflächenelementen trennt. Dieser Abschnitt soll vor allem den Unterschied zwischen den Oberflächen-Elementen und den dem Entwickler zur Verfügung stehenden Instanzen der jeweiligen Kapsel-Klassen verdeutlichen.

Bei folgender häufig verwendeten Ereignisprozedur eines Formulars sind beispielsweise drei verschiedene Objekte beteiligt:

```
Private Sub cmdClose_Click()
   DoCmd.Close acForm, Me.Name
End Sub
```

Me

`Me` ist der Name einer Referenz auf das Objekt der Formularklasse des vorliegenden Formulars. Der Name der Formularklasse ist `Form_<Formularname>`. Hat das Formular beispielsweise den Namen `frmKunden`, so ist der Name der von Access generierten Formular-Klasse `Form_frmKunden`. Hingegen ist `Me` die Referenz auf jene Instanz der Formularklasse, die das vorliegende Formular kapselt. Die spezielle Formularklasse (wie das genannte Beispiel `Form_frmKunden`) basiert auf der generischen Access-Formularklasse `Form`. Diese generische Klasse wird durch folgende Elemente erweitert:

- Die Felder der zugrunde liegenden Datensatzherkunft (Eigenschaft `RecordSource`)
  Für jedes Feld der zur Entwurfszeit bekannten Datensatzherkunft wird eine Eigenschaft angelegt (Verborgener Datentyp `AccessField` aus der Bibliothek `Access`, einziges Element ist die Standardeigenschaft `Value`).
- Die Steuerelemente
  Für jedes eingefügte Steuerelement wird eine gleichnamige Eigenschaft erzeugt. Der Datentyp entspricht der jeweiligen Kapselklasse.

Das Bewusstsein über diese Zusammenhänge bietet interessante Möglichkeiten für die Access-Praxis. Elemente einer benutzerdefinierten Klasse können auch dem Objektmodul eines Formulars oder Berichts hinzugefügt werden. So kann ein Kunden-Formular zum Beispiel mit einer Eigenschaft `KundenID` ausgestattet werden. Durch entsprechende Programmierung kann nun einfach durch Zuweisen eines gültigen Primärschlüsselwertes an die Eigenschaft `KundenID` der entsprechende Datensatz im Formular angezeigt werden.

Beispiel:
Im Objektmodul von frmKunden:

```
Public Property Get KundenID(ByVal NewKundenID As Long)
   mp_lngKundenID = NewKundenID
   Me.RecordSource = "SELECT * FROM tblKunden " & _
                     "WHERE KundenID = " & mp_lngKundenID
End Property
```

Ändern des angezeigten Datensatzes in frmKunden von außerhalb:

```
Forms("frmKunden").KundenID = ...
```

Anmerkung: Da es sich bei der Forms-Auflistung um eine polymorphe Auflistung handelt, versagt bei dieser Syntax Intellisense seinen Dienst was selbst geschriebene Elemente wie die genannte Eigenschaft betrifft. Allen Referenzen der Form `Forms("<Formularname>")` sind jedoch die Elemente der Klasse `Access.Form` eigen, weshalb von Intellisense auch nur diese vorgeschlagen werden.
Wird hingegen ein korrekt typisierter Objektverweis verwendet, klappt es auch mit Intellisense wieder:

```
Dim frmKunden As Form_frmKunden

Set frmKunden = Forms("frmKunden")
frmKunden.KundenID = ...  ' Hier funktioniert auch Intellisense!
Set frmKunden = Nothing
```

cmdClose

Das Objekt `cmdClose` kapselt den Zugriff auf das Schaltflächen-Steuerelement cmdClose. Wenn der Entwickler in der Entwurfsansicht des Formulars eine Schaltfläche zeichnet, erzeugt Access zusätzlich nicht sichtbaren Code zur Erstellung einer Instanz der jeweiligen Steuerelement-Klasse. Der Name dieser Instanz ist gleich dem Namen des Steuerelements. Die verwendete Steuerelement-Klasse hängt vom Steuerelement ab. Die Klasse für Befehlsschaltflächen ist beispielsweise `CommandButton` aus der Bibliothek `Access`. Das Vorhandensein dieses Steuerelement-Objekts zeigt sich auch dadurch, dass in der Objekt-Liste des VBE der Name jedes Steuerelements aufscheint. In der rechten Liste des VBE finden sich alle Ereignisse des in der Objekt-Liste ausgewählten Steuerelements, oder genauer gesprochen des Steuerelement-Objekts.
Im vorliegenden Beispiel wird auf das Ereignis Click dieses Steuerelements reagiert (Ereignisprozedur `cmdClose_Click`).

DoCmd

Das `DoCmd`-Objekt dient zum Ausführen von Access-Aktionen von Visual Basic aus. Das `DoCmd`-Objekt ist eine immer verfügbare Instanz der Klasse `DoCmd` aus der Bibliothek `Access`. Das `DoCmd`-Objekt wird über die gleichnamige Eigenschaft des `Application`-Objekts zur Verfügung gestellt. Es ist nicht möglich, zusätzliche Instanzen der Klasse `DoCmd` zu erstellen.

Einige Methoden des `DoCmd`-Objekts lassen sich durch objektbasierte Pendants ersetzen. So lässt sich das Speichern des aktuellen Datensatzes sowohl durch

```
DoCmd.RunCommand acCmdSaveRecord
```

also auch durch

```
Me.Dirty = False
```

erreichen.

Der Vorteil der zweiten Variante ist, dass das gemeinte Formular explizit angesprochen wird. Durch die Verwendung der Eigenschaft Dirty des jeweiligen Formular-Objekts ist es nun beispielsweise auch möglich, geänderte Daten in mehreren Formularen zu speichern, ohne den Fokus zwischen Formularen zu wechseln.

Beispiel:

```
Forms("frmKunden").Dirty = False
Forms("frmBestellungen").Dirty = False
```

Entwickeln benutzerdefinierter Klassen in VB(A)

Elemente

Konstruktion und Destruktion

Schnittstellen

Auflistungs-Klassen

Allgemeines

Zum Erstellen einer benutzerdefinierten Klasse in Access dienen Klassenmodule. In Access 97 kann ein leeres Klassenmodul bei aktiviertem Datenbank-Fenster über den Menüpunkt Einfügen-Klassenmodul eingefügt werden. Ab Access 2000 stehen zusätzlich im VBE verschiedene Möglichkeiten zur Verfügung. Erwähnen möchte ich hier vor allem das Kontext-Menü des Projekt-Explorers (rechte Maustaste im Projekt-Explorer, Einfügen – Klassenmodul).

Jede Klasse benötigt ein eigenes Klassenmodul. Der Name des Klassenmoduls bestimmt auch den Namen der Klasse. Der Name kann im Datenbank-Fenster oder im Eigenschaften-Toolfenster des VBE geändert werden.

Elemente

Im Folgenden wird beschrieben, wie die Elemente einer benutzerdefinierten Klasse (Eigenschaften, Methoden und Ereignisse) erzeugt werden können.

Eigenschaften

Zum Erstellen von Eigenschaften stehen zwei Varianten zur Verfügung – die Erstellung eines Feld (einer öffentlichen Modulvariablen) oder die Erstellung von Property -Funktionen bzw. -Prozeduren.

Öffentliche Modulvariable

Dies stellt die einfachste und gleichzeitig auch eingeschränkteste Art der Implementierung einer Eigenschaft dar.

Beispiel (Eigenschaft LastName)
Erstellung (in Klassenmodul `CMitarbeiter`):

```
Public LastName As String
```

Zugriff:

```
Dim objChef As CMitarbeiter

Set objChef = New CMitarbeiter

objChef.LastName = "Mayer"    ' Schreib- und Lesezugriff direkt ...
MsgBox objChef.LastName    ' ... über die Modulvariable

Set objChef = Nothing
```

Nachteil:

- **Unkontrollierter Zugriff** von außen
- Name des Feldes ist nach außen hin sichtbar. Benennungskonventionen können nicht eingehalten werden.

Vorteil:

- Sehr einfach zu implementieren

Eigenschafts-Prozedur

Für den Schreib- und Lesezugriff stehen getrennte Prozeduren bzw. Funktionen, sogenannte Accessoren, zur Verfügung.

Lesezugriff:	`Property Get`
Schreibzugriff:	`Property Let` bei Werttypen bzw. `Property Set` bei Referenztypen

Die Speicherung des Werts der Eigenschaft erfolgt sehr häufig in einer privaten Member-Variablen.

Typisches Beispiel (Implementierung Eigenschaft `LastName` im Klassenmodul `CMitarbeiter`):

```
Private mp_strLastName As String

Public Property Let LastName(NewLastName As String)
   mp_strLastName = NewLastName
End Property
Public Property Get LastName() As String
   LastName = mp_strLastName
End Property
```

Zugriff (wie zuvor):

```
Dim objChef As CMitarbeiter

Set objChef = New CMitarbeiter

objChef.LastName = "Mayer"    ' Schreibzugriff über Property Let
MsgBox objChef.LastName       ' Lesezugriff über Property Get

Set objChef = Nothing
```

Wird der Wert der Eigenschaft gesetzt, so erfolgt ein Aufruf der Eigenschafts-Prozedur Property Let, bei der der neue Wert als letzter (hier: einziger – siehe unten) Parameter übergeben wird. In der Prozedur erfolgt hier lediglich das Durchreichen an die private Modulvariable.
Wird der Wert der Eigenschaft gelesen, so erfolgt ein Aufruf der Eigenschafts-Funktion Property Get. Der Rückgabewert dieser Funktion definiert den von der Eigenschaft zurückgegebenen Wert. In der hier gezeigten einfachen Implementierung wird wiederum nur der Wert der privaten Modulvariable zurückgegeben.

Nachteil:

- **Aufwändiger** zu implementieren

Vorteile:

- **Kontrollierter Zugriff**
  Bei einem Schreibzugriff kann in der Eigenschafts-Prozedur Property Let bzw. Property Set die Einhaltung von Grenzen überprüft und durchgesetzt werden. Bei Zuweisung eines Wertes außerhalb des gültigen Bereichs kann der Wert auf den gültigen Bereich beschränkt werden oder ein Fehler, wie im einfachsten Fall der allgemeine Fehler 5 ("Ungültiger Prozeduraufruf oder ungültiges Argument"), aufgeworfen werden.

```
Public Property Let Monat(ByVal NewMonat As Byte)
   If NewMonat < 1 Then NewMonat = 1
   If NewMonat > 12 Then NewMonat = 12
   mp_bytMonat = NewMonat
End Property
```

- Objekt kann **auf Änderungen reagieren**

```
Public Property Let ParentID(NewParentID As Long)
   If NewParentID <> mp_lngParentID Then
      Call m_objParent.Detach(Me)
      mp_blnDirty = True
      mp_lngParentID = NewParentID
   End If
End Property
```

- **Berechnete Werte** (Saldo, PrettyName, Anzahl, ...)
- **Schreibgeschützte Eigenschaften** (kein Let bzw. Set)

```
' Keine Property Let/Set-Prozedur -> Read only
Public Property Get PrettyName() As String
   PrettyName = mp_strLastName
   If Len(mp_strFirstName) > 0 Then
      PrettyName = PrettyName & " " & mp_strFirstName
   End If
End Property
```

- **Write-Once**-Eigenschaften (einmalige Initialisierung)

```
Public Property Let CustomerID(NewCustomerID As Long)
   If mp_lngCustomerID = 0 Then
      mp_lngCustomerID = NewCustomerID
   Else
      Err.Raise vbObjectError + 1234
   End If
End Property
```

Es ist möglich, die Sichtbarkeit der Teilprozeduren eines zusammengehörenden Let/Get- bzw. Set/Get-Paares unterschiedlich zu gestalten. So kann die Property Let-Prozedur privat (`Private`) und die Property Get-Funktion öffentlich (`Public`) gemacht werden. Damit ist die Eigenschaft für den Entwickler der Klasse les- und schreibbar, für einen Verwender der Klasse jedoch schreibgeschützt.

Unterschiede bei Wert- und Referenztypen

Je nachdem, ob der Datentyp der zu erstellenden Eigenschaft ein Wert- oder Referenztyp ist, sind kleine Unterschiede zu beachten:

Werttypen

Die Zuweisung an die Eigenschaft erfolgt über eine `Property Let`-Eigenschafts-Prozedur. Die Parameterübergabe kann per Wert (`ByVal`) erfolgen:

```
Public Property Let Name(ByVal NewName As String)
   mp_strName = NewName
End Property
Public Property Get Name() As String
   NewName = mp_strName
End Property
```

Verwendung ohne Set:

```
objCustomer.Name = "Mayer"
```

Referenztypen

Die Zuweisung an die Eigenschaft erfolgt über eine Property Set-Eigenschafts-Prozedur. Die **Parameterübergabe** muss **per Referenz** erfolgen (Standard in VB-Classic). Set-Anweisungen sind auch bei der Angabe des Rückgabewertes der Property Get-Eigenschafts-Funktion erforderlich:

```
Public Property Set Parent(ByRef NewParent As Access.Form)
   Set mp_frmParent = NewParent
End Property
Public Property Get Parent() As Access.Form
   Set Parent = mp_frmParent
End Property
```

Verwendung mit Set:

```
Set objListener.Parent = Me
```

Parametrisierte Eigenschaften

Eigenschafts-Prozeduren und -Funktionen können auch Parameter mitgegeben werden, wodurch sich parametrisierbare Eigenschaften erstellen lassen. Diese Eigenschaften können optional sein. Bei Property Let/Set-Eigenschafts-Prozeduren gibt der letzte Formalparameter immer den neuen Wert der Eigenschaft an.

Konsistenz von Property Let/Set und Property Get

Die Parameterliste einer Property Get-Eigenschafts-Funktion kann wie die jeder anderen Funktion gestaltet sein. Insbesondere können die letzten Parameter als Optional gekennzeichnet sein.

Die Paramterliste der zugehörigen Property Let bzw. Property Set-Eigenschafts-Prozedur muss bezüglich der Datentypen genau jener der Property Get-Prozedur mit hinten angehängtem zusätzlichen Parameter zur Wertübergabe entsprechen.

Im Falle optionaler Parameter kommt es dadurch zum sonst syntaktisch nicht erlaubten Fall, dass hinter einem optionalen Parameter ein nicht optionaler Parameter folgt.

Column-Eigenschaft eines Access-Kombinationsfeldes

Ein Beispiel für die Anwendung einer parametrisierten Eigenschaft mit optionalem Parameter in Access ist die Eigenschaft Column des Kombinationsfeld- und Listenfeld-Steuerelements. Die Parameter sind Col und Row, wobei letzterer optional ist.

Die Rümpfe der beiden Accessoren mit gleicher Signatur müssten in einer benutzerdefinierten Klasse daher folgendermaßen definiert werden:

```
Public Property Column Let(Col As Long, Optional Row As Long, _
                           ByVal NewColumn As String)
   ' ...
End Property
Public Property Column Get(Col As Long, Optional Row As Long)
   ' ...
End Property
```

Der Einsatzbereich parametrisierter Eigenschaften sind z.B. indizierte Objekte oder assoziative Arrays.

Methoden

Die Methoden erstellt man einfach als (öffentliche) **Funktionen** und **Prozeduren** der Klasse. Innerhalb der Methoden ist der Zugriff auf private Member möglich.

Beispiel:

```
Public Function GetBalance(ByVal AccYear As Integer) As Currency
   GetBalance = m_curHaben - m_curSoll
End Function
```

Zugriff:
Methode ohne Rückgabewert (Public Sub)

```
Call MyCustomer.PrintOut(StdPrinter)
```

Methode mit Rückgabewert (Public Function)

```
curSaldoVorjahr = MyAccount.GetBalance(Year(Now()) - 1)
```

Private Funktionen und Prozeduren können natürlich als Hilfsfunktionen innerhalb der Klasse verwendet werden.

Ereignisse

Benutzerdefinierte Klassen können erst **ab Access 2000** Ereignisse auslösen. Ereignisse müssen zuerst mit der `Event`-Anweisung deklariert werden, bevor sie zum entsprechenden Zeitpunkt mit der `RaiseEvent`-Anweisung ausgelöst werden können.
Deklaration:

```
Public Event <Ereignis>([<Parameterliste>])
```

Auslösung:

```
RaiseEvent <Ereignis>([<Parameterliste>])
```

Beispiel:

```
' Im Deklarationsabschnitt:
Public Event SizedControl(ByRef ctrl As Access.Control)

' ...

' In einer Funktion oder Prozedur
   RaiseEvent SizedControl(ctlCurrent)
```

Rückmeldungen an das Objekt

Mit Hilfe von Referenz-Parametern lässt sich ein Mechanismus zur Rückmeldung von der Ereignis-Prozedur an das das Ereignis auslösende Objekt erreichen. Dieser Mechanismus wird in Access immer wieder bei Cancel-Parametern verwendet (z.B. `Form_Open`). Ein anderes Beispiel ist der Response-Paramter der Ereignisse `BeforeDelConfirm` und `Error` des Formular-Objekts.

Beispiel Lösch-Vorgang in Klasse `CCustomers`:

In Klassenmodul `CCustomers`:

```
Public Event DeleteCustomer(ByRef Customer As CCustomer, _
                            ByRef Cancel As Boolean)

' ...

Private Sub Delete(Customer As CCustomer)
   Dim blnCancel As Boolean

   blnCancel = False
   RaiseEvent DeleteCustomer(Customer, blnCancel)
   if Not blnCancel Then
      ' ... Kunden löschen
   End If
End Sub
```

In der übergeordneten Klasse:

```
Private WithEvents MyCustomers As CCustomers

' ...

Private Sub MyCustomers_DeleteCustomer( _
                           ByRef Customer As CCustomer, _
                           ByRef Cancel As Boolean)
   Cancel = (vbCancel = _
             MsgBox("Der Kunde " & Customer.FullName & " " & _
                    "wird nun endgültig gelöscht.", _
                    vbOkCancel + vbExclamation))
End Sub
```

Konstruktion und Destruktion

Bei einem Objekt sind der Zeitpunkt des Beginns seiner Lebensdauer als auch des Endes seiner Lebensdauer von Bedeutung. Diese Vorgänge nennt man Konstruktion bzw. Destruktion.

Die Ereignisse Initialize und Terminate

Wird ein Objekt im Speicher erzeugt, so tritt das Klassenereignis `Initialize` auf (Ereignis-Prozedur `Class_Initialize`). Beim Zerstören des Objekts tritt das Klassenereignis `Terminate` auf (Ereignis-Prozedur `Class_Terminate`). Code in diesen beiden Prozeduren wird beim Erzeugen bzw. Zerstören eines Objekts aufgerufen. In `Class_Initialize` werden üblicherweise Initialisierungen der internen Variablen vorgenommen. In `Class_Terminate` können die während der Lebensdauer des Objekts angeforderten Ressourcen wieder freigegeben werden.

Beide Methoden sind nicht parametrisierbar (im Gegensatz zu Konstruktoren in C++, Java, VB.NET, C#, ...). Es ist also nicht möglich, bereits direkt am Beginn der Lebensdauer des Objekts Informationen zu übermitteln.

Dies muss nach dem eigentlichen Instanziieren des Objekts durch Zuweisung an die entsprechenden Eigenschaften oder durch Aufruf einer Intialisierungs-Methode erfolgen. Es besteht daher die Gefahr, dass ein nicht ausreichend initialisiertes Objekt verwendet wird.

Eine Abhilfe dafür ist die Bereitstellung einer sogenannten Factory-Methode oder eines Factory-Objekts. Dieser Methoden werden die zur Initialisierung des Objekts nötigen Parameter übergeben. Sie übernimmt die eigentliche Instanziierung und Initialisierung und liefert eine Referenz auf das neue Objekt zurück.

Beispiel:
Annahme: Jedes Objekt der Klasse CCustomer benötigt zumindestens eine CustomerID und eine KundenNr (erforderliches Feld):
Problemszenario:

```
Public Sub FehlerhafteVerwendung(KundenNr As String)
   Dim cstNew As CCustomer

   Set cstNew = New CCustomer

   ' Unzureichende Initialisierung, KundenNr fehlt!
   cstNew.CustomerID = GetNextCustomerID()
   cstNew.Save           ' Fehler erst zur Laufzeit!

   Set cstNew = Nothing
End Sub
```

Lösung mittels Factory-Methode:

```
Public Function CreateCustomer(NewCustomerID As Long, _
                               NewKundenNr As String) _
                As CCustomer
   Dim cstNew As CCustomer

   Set cstNew = New CCustomer
   With cstNew
      .CustomerID = NewCustomerID
      .KundenNr = NewKundenNr
   End With

   Set CreateCustomer = cstNew

   Set cstNew = Nothing
End Function
```

Verwendung der Factory-Methode:

```
Public Sub KorrekteVerwendung(KundenNr As String)
   Dim cstNew As CCustomer

   Set cstNew = CreateCustomer(GetNextCustomerID(), _
                               KundenNr)

   cstNew.Save           ' Kein Problem

   Set cstNew = Nothing
End Sub
```

Schnittstellen (Interfaces)

Zur Erstellung einer Schnittstelle dient ein Klassenmodul. In ihm befinden sich nur die Deklarationen der Elemente der Schnittstelle, aber im Allgemeinen keine Implementierungen. Da VB keine Implementierungsvererbung unterstützt, sind Implementierungen in Schnittstellen für Klassen, die die Schnittstelle implementieren ohnehin nicht verwendbar.

Die Implementierung der Schnittstelle, d.h. das Schreiben des Codes, der die tatsächliche Funktionalität bereitstellt, erfolgt in der jeweiligen Klasse. Mit dem Schlüsselwort Implements wird in der implementierenden Klasse die zu implementierende Schnittstelle angegeben:

```
Implements <Schnittstelle>
```

Implementierung einer Eigenschaft

Die Implementierung einer Eigenschaft `<Eigenschaft>` der Schnittstelle `<Schnittstelle>` erfolgt durch Bereitstellen von Eigenschaften-Prozeduren mit dem Namen `<Schnittstelle>_<Eigenschaft>`:

```
Private Property Let|Set _
      <Schnittstelle>_<Eigenschaft>([<Parameterliste>], _
                                    NewValue As <Datentyp>)
   ' Konkrete Implementierung
End Property
Private Property Get _
      <Schnittstelle>_<Eigenschaft>([<Parameterliste>]) _
      As <Datentyp>
   ' Konkrete Implementierung
End Property
```

Öffentliche Modulvariablen der Schnittstelle[5] müssen in der implementerenden Klasse durch ein Paar von `Let/Get-` bzw. `Set/Get`-Eigenschaften-Prozeduren implementiert werden. Schreib- und lesegeschützte Eigenschaften sind möglich.

Implementierung einer Methode

Die Implementierung einer Methode `<Methode>` der Schnittstelle `<Schnittstelle>` erfolgt durch Bereitstellen einer Methode `<Schnittstelle>_<Methode>`:

```
Private Sub|Function _
   <Schnittstelle>_<Methode>([<Parameterliste>]) _
   [As <Datentyp>]
   ' Konkrete Implementierung
End Sub|Function
```

Für eine erfolgreiche Kompilierung des Programms ist die vollständige Implementierung der Schnittstelle notwendig.

Es ist üblich, den Namen einer Schnittstelle mit `I` zu beginnen (z.B. `IMitarbeiter`).

[5] Private Elemente sind nicht Teil der Schnittstelle und daher in einer Interface-Definition sinnlos.

Beispiel

Es existiert eine Klasse zum Sortieren (`CSorter`). Ihre Implementierung basiert ausschließlich auf Schnittstellen.
Die Signatur der zentralen Methode `Sort` sieht folgendermaßen aus:

```
Sub Sort(coll As ISortableCollection, comp As IComparer)
```

Es lassen sich damit also Objekte jeder Klasse, die die Schnittstelle `ISortableCollection` implementiert unter Zuhilfenahme eines beliebigen Vergleicher-Objekts, dessen Klasse lediglich die Schnittstelle `IComparer` implementiert vergleichen.
In der Schnittstelle `IComparer` ist nun die Signatur für die Methode `Compare` definiert:

```
Public Function Compare(o1 As IBusinessObject, _
                        o2 As IBusinessObject) As Long
End Function
```

Die Implementierung der Methode muss also zwei Objekte erwarten, deren Klassen die Schnittstelle `IBusinessObject` implementieren, und liefert den Vergleichswert zurück.

Im Folgenden sehen wir eine Implementierung dieser Schnittstelle `IComparer` in der Klasse `CCompareCustomerByLastName`:

```
Implements IComparer

Private Function IComparer_Compare(o1 As IBusinessObject, _
                                   o2 As IBusinessObject) _
                 As Long
   Dim c1 As CCustomer
   Dim c2 As CCustomer
   Set c1 = o1
   Set c2 = o2
   IComparer_Compare = StrComp(c1.LastName, c2.LastName)
   Set c2 = Nothing
   Set c1 = Nothing
End Function
```

Ein Objekt dieser Klasse kann nun an die `Sort`-Methode der `CSorter`-Instanz übergeben werden:

```
Dim objSorter As CSorter
Dim cc as CCustomers
' ...
Set objSorter = New CSorter
Call objSorter.Sort(cc, New CCompareCustomerByLastName)
Set objSorter = Nothing
Set cc = Nothing
```

Für eine komplexere Anwendung der Sortier-Klasse siehe “Datenkapsel-Klassen” auf Seite 44.

Auflistungs-Klassen

In VB können eigene Klassen als Auflistungsklassen inklusive der Unterstützung von **For Each**-Schleifen definiert werden.

Im Wesentlichen besteht die Implementierung einer eigenen Auflistungsklasse in der Kapselung des direkten Zugriffs auf die Elemente einer Standard-VB(A)-Collection sowie in der Bereitsstellung spezifischer Zusatzfeatures (z.B. Laden aus einer Datenbank-Tabelle).

For Each-Unterstützung

Zur Unterstützung von For Each-Schleifen muss eine Methode oder Eigenschaft eine Referenz auf einen sogenannten Enumerator in Form einer `IUnknown`-Referenz zurückliefern. Dieser Wert wird direkt aus der privaten Standard-Collection übernommen. Er ist dort unter dem versteckten Namen `_NewEnum` vorhanden. Da Namen in VB(A) nicht mit einem Unterstrich beginnen dürfen, muss in VB(A) die Syntax `[_NewEnum]` verwendet werden:

```
Public Property Get NewEnum() As IUnknown
   Set NewEnum = m_coll.[_NewEnum]
End Property
```

Damit diese Eigenschaften-Prozedur aber nun auch vom VB-internen Mechanismus der `For-Each`-Schleife erkannt wird, muss noch die Prozedur-ID der Prozedur auf den Wert –4 gesetzt werden. Der Name der Eigenschaft (hier `NewEnum`) ist jedoch nicht relevant. Dieser Vorgang wird von der VBA-IDE jedoch nicht unterstützt. Durch einen kleinen Trick kann das Problem aber umgangen werden (siehe unten).

Standard-Element

Bei Auflistungsklassen ist meist die Eigenschaft `Item` das Standard-Element der Klasse. Um dies auch in der eigenen Auflistungsklasse zu erreichen, muss auch für diese Methode eine Prozedur-ID angegeben werden. Ihr Wert ist 0. Auch hier kann der unten angeführte Trick zur Anwendung kommen.

Beispiel einer Customers-Klasse (Einfachst-Variante):

```
Private m_coll As Collection

Private Sub Class_Initialize()
   Set m_coll = New Collection
End Sub

Private Sub Class_Terminate()
   Set m_coll = Nothing
End Sub

Public Property Get Count() As Long
   Count = m_coll.Count
End Property

Public Function Add(c as CCustomer, _
                    Optional Key As String) As CCustomer
   If Len(Key) = 0 Then
      m_coll.Add c
   Else
      m_coll.Add c, Key
   End If
   Set Add = c
End Function

Public Sub Remove(Index As Variant)
   m_coll.Remove Index
End Sub

Public Function Item(Index As Variant) As CCustomer
Attribute Item.VB_UserMemId = 0   ' Standard-Element (optional)
                                  ' muss in der VBA-IDE mittels
                                  ' Trick eingefügt werden
   Set Item = m_coll(Index)
End Function

Public Property Get NewEnum() As IUnknown
Attribute NewEnum.VB_UserMemId = -4    ' Enumerator muss in
                                       ' der VBA-IDE mittels
                                       ' Trick eingefügt werden
   Set NewEnum = m_coll.[_NewEnum]
End Property
```

Trick zum Erzeugen der Attribute in der VBA-IDE

Der Code, den man in der IDE zu Gesicht bekommt, ist nicht der vollständige Code, aus dem die Klasse besteht. Zusätzlich zu den in der IDE sichtbaren Zeilen gehören noch einige Meta-Informationen, die teilweise über die Eigenschaften-Toolbox der IDE eingestellt werden können (z.B. "Instancing").

Weitere Meta-Informationen sind die sogenannten Attribute, über die auch die Prozedur-IDs für den Enumerator und das Standardelement eingestellt werden.

Im kompletten Quellcode einer Klasse sehen diese Zeilen wie folgt aus:

```
Attribute Item.VB_UserMemId = 0     ' Element Item als Standard-
                                    ' element kennzeichnen
Attribute NewEnum.VB_UserMemId = -4 ' Element NewEnum als
                                    ' Enumerator kennzeichnen
```

Die VB(6)-IDE bietet zur Einstellung dieser Attribute das Dialogfeld "Prozedur-Attribute". In der VBA-IDE fehlt aber dieses Feature. Es kann jedoch das Klassenmodul in eine Textdatei exportiert werden, das Attribut mit einem Text-Editor eingefügt und die geänderte Datei danach wieder importiert werden.

Mit einem kleinen Trick können die Arbeitsschritte dennoch auf die IDE beschränkt bleiben:

- In der IDE das Attribut als erste Zeile in der Prozedur einfügen. Diese Zeile ist keine gültige Quellcode-Zeile und wird daher vom Compiler als syntaktisch falsch bemängelt. Ignorieren Sie diesen Fehler.
- Das Klassenmodul in eine Textdatei exportieren (im Projekt-Explorer: rechte Maustaste auf das Klassenmodul, Eintrag "Entfernen von <Modulname>", Exportieren: "Ja" (beliebiger Pfad)
- Das exportierte Klassenmodul wieder einfügen (im Projekt-Explorer: rechte Maustaste, Eintrag "Datei importieren...", zuvor exportierte Datei auswählen. – Fertig!

Reaktion auf Ereignisse von Access-Objekten

Mit Hilfe des Schlüsselwortes `WithEvents` kann auch auf das Eintreten eines Ereignisses eines Objekts reagiert werden. Zur Illustration möchte ich hier noch einmal das Beispiel von Seite 29 anführen:

```
Private WithEvents MyCustomers As CCustomers

' ...

Private Sub MyCustomers_DeleteCustomer( _
                          ByRef Customer As CCustomer, _
                          ByRef Cancel As Boolean)
   ' ...

End Sub
```

Zur erfolgreichen Implementierung eines Ereignis-Handlers ist also zum einen die Deklaration der Objektvariablen MyCustomers mit dem Schlüsselwort `WithEvents` nötig und zum anderen die Bereitstellung der Ereignis-Routine `MyCustomers_DeleteCustomer(...)`.

Bei Access-Objekten wie Formularen oder Steuerelementen ist diese Struktur jedoch um einen wesentlich Punkt zu erweitern wie wir sofort sehen werden. Im Sinne des obigen Code-Beispiels könnte man folgendes versuchen um innerhalb eines benutzerdefinierten Objekts auf das Click-Ereignis der Schaltfläche `cmdClose` eines Formulars zu reagieren:

Im Klassenmodul `CEventSink`:

```
Private WithEvents mp_cmdClose As Access.CommandButton

Public Property Set CmdClose(ByRef NewCmdClose _
                             As Access.Command)
   Set mp_cmdClose = NewCmdClose
End Property

Private Sub mp_cmdClose_Click()
   ' Dieser Code sollte beim Klicken auf die Schaltfläche
   ' ausgeführt werden. Wird er aber wahrscheinlich nicht.
End Sub
```

Im Formular:

```
Private m_es As CEventSink

Private Sub Form_Open(ByRef Cancel As Integer)
   Set m_es = New CEventSink
   Set m_es.CmdClose = Me.cmdClose
End Sub

Private Sub Form_Close()
   Set m_es = Nothing
End Sub
```

Die Code-Abschnitte wirken auf den ersten Blick korrekt. Mittels der Eigenschaft `CmdClose` wird an das Objekt `m_es` eine Referenz auf das Schaltflächen-Objekt `cmdClose` weitergereicht. Die Eigenschafts-Prozedur setzt nun auch die mit `WithEvents` deklarierte Member-Variable `mp_cmdClose` des Klassenmoduls auf diese Referenz. Warum wird dennoch das Ereignis wie im ersten Code-Abschnitt angeführt wahrscheinlich nicht die Ereignisprozedur auslösen?
Zur Beantwortung dieser Frage werfen wir einen Blick auf die Art und Weise, wie Access mit Ereignissen umgeht.

> Beachten Sie hier auch bitte unbedingt den Access-Bug in Bezug auf WithEvents (“Ein Access-Bug” auf Seite 39).

Ereignis-Struktur bei Access-Oberflächen-Objekten

Auf den Eintritt eines Ereignisses eines Access-Oberflächen-Objekts kann auf drei verschiedene Arten reagiert werden.
Es kann

- ein Makro aufgerufen werden
- eine VBA-Funktion aufgerufen werden
- eine Ereignis-Prozedur ausgelöst werden.

Die Steuerung, welche der drei Varianten zum Einsatz kommt, obliegt der jedem Ereignis zugeordneten Ereignis-Eigenschaft. Diese Ereignis-Eigenschaften sind im Eigenschaften-Fenster des jeweiligen Objekts sicht- und änderbar:

Hier ist nun auch schon zu sehen, was vorhin das Wort "wahrscheinlich" gerechtfertigt hat. Steht aufgrund einer früher schon eingetragenen Ereignis-Prozedur der Wert der Ereignis-Eigenschaft zufällig auf `"[Ereignisprozedur]"`, so wird auch obiger Code bereits die erwartetete Reaktion gezeigt haben. Andernfalls wird die Ereignis-Prozedur in der Klasse `CEventSink` jedoch nie ausgelöst werden, da die Verbindung zwischen dem Ereignis und den VB-Ereignis-Handlern nicht besteht.

Die Lösung besteht nun darin, den Wert der Ereignis-Eigenschaft auf den notwendigen Wert zu stellen. Dies kann zum Glück per Code aus unserer Klasse erledigt werden. Die Ereignis-Eigenschaften haben meistens den Namen `On<Ereignis>`. Die Ereignis-Eigenschaft für das Click-Ereignis heißt beispielsweise `OnClick`. Den exakten Namen findet man am leichtesten durch Drücken von F1 wenn die gewünschte Eigenschaft im Eigenschaften-Fenster den Fokus hat.

Bei der Einstellung dieser Ereignis-Eigenschaft ist zu beachten, dass vom VB-Code aus alle Bezeichnung nicht lokalisert sind, sondern in Englisch eingetragen werden müssen. Der korrekte Eintrag für die Behandlung eines Ereignisses mittels Event-Handler lautet `"[Event Procedure]"`.

Die korrigierte Version der Klasse `CEventSink` sieht also nun so aus:

```
Private WithEvents mp_cmdClose As Access.CommandButton

Public Property Set CmdClose(ByRef NewCmdClose As Access.Command)
   Set mp_cmdClose = NewCmdClose
   mp_cmdClose.OnClick = "[Event Procedure]"
End Property

Private Sub mp_cmdClose_Click()
   ' Dieser Code sollte beim Klicken auf die Schaltfläche
   ' ausgeführt werden. Wird er aber wahrscheinlich nicht.
End Sub
```

Benutzerdefinierte Klassen im Praxiseinsatz

Helfer

Compound-Controls

Datenhelfer

Datenkapsler

Bemerkung zu Beginn

Alle im Folgenden vorgestellten Klassen befinden sich im Real-World-Einsatz in Projekten des Autors. Die zum Erstellen solcher Klassen notwendigen Grundlagen sind in diesem Skriptum verzeichnet. Wie immer steckt jedoch auch hier der Teufel im Detail. Gerade im Zusammenhang mit der Ereignis-Verarbeitung unter Access stellt sich eine geniale, auf einem einfachen Mechanismus beruhende Idee, beim Versuch einer Projekt-tauglichen Implementierung sehr gern als äußerst mühsames Unterfangen heraus.

Die angegebenen Code-Beispiele zeigen jeweils nur eine exemplarische Anwendung der jeweiligen Klassen. Dies soll demonstrieren, mit welch geringem Aufwand das Hereinholen von zusätzlicher Funktionalität verbunden ist.

Ein Access-Bug

Den Praxiseinsatz von benutzerdefinierten Klassen in Access zu beschreiben, ohne auf den unsäglichen Bug in diesem Zusammenhang hinzuweisen wäre wohl fahrlässig. Daher hier der folgende Hinweis:

> Wenn Sie versuchen ein Formular zu schließen, das die Referenz auf ein Objekt hält, kann Access abstürzen oder eine Fehlermeldung produzieren (Microsoft Knowledge Base).
>
> Nach meiner Erfahrung betrifft dies nur Objekte, die ihrerseits eine mit `WithEvents` deklarierte Referenz auf ein Formular-Objekt halten, wenn man versucht, die Referenz auf das Objekt beim Beenden des Formulars auf `Nothing` zu setzen.

Dies ist ein bekannter und in der Knowledge-Base (Artikel 223245) dokumentierter Bug, der in allen mir vertrauten Access-Versionen (A97, A2k und A2k2) auftritt.

Zu Umgehung dieses Themas gibt es zwei Zugänge:

1. Sie verzichten auf das eigentlich korrekte Zuweisen der Objektreferenz auf `Nothing`. Dadurch wird aber das Objekt nicht korrekt deinstanziiert. Bei oftmaligem Öffnen und wieder Schließen des Formulars bedeutet dies ein immer größer werdendes Speicherleck. Da Access-Anwendungen kaum 24h und sieben Tage die Woche laufen werden, stellt das wahrscheinlich kein wirklich großes Problem dar. Unschön bleibt die Sache jedoch allemal.
2. Bei dem Problem dürfte es sich zumindestens in manchen Situationen um ein Timing-Problem handeln. Folgende Vorgangsweise kann daher manchmal zu Ziel führen:

   Öffnen Sie im `Close`-Event des Formulars versteckt ein weiteres Formular in dem Sie einen Timer mit sehr kurzem Intervall (einige Millisekunden reichen meist) laufen haben und übergeben Sie diesem Formular eine Referenz auf das fragliche Objekt. Im `Timer`-Ereignis dieses Formulars können Sie nun (mit etwas Glück – sic!) die übergebene Referenz auf `Nothing` setzen und das versteckte Formular auch wieder schließen. Damit sollte sich auch das Objekt aus dem Speicher entfernen.

Helfer-Klassen (nicht Daten-gebunden)

CNamedValues

- Erlaubt **Serialisierung/Deserialisierung** von mehreren Namen/Wert-Kombinationen in einen/aus einem **String**
- Objektorientiertes Interface
- Anwendung: **Übergabe mehrerer Werte in OpenArgs**
- Ähnlich CTaggedValues aus dem ADH (mit Erweiterungen)

Beispiel

Im aufrufenden Formular:

```
Dim nv As CNamedValues

Set nv = New CNamedValues
With nv
   .Item("JustHideOnOk") = True
   .Item("AutoSave") = False
   If Not IsNull(Me.KundenID.Value) Then
      .Item("KundenID") = Me.KundenID.Value
   End If
   DoCmd.OpenForm "frmKunden", OpenArgs:=.AsString
End With
Set nv = Nothing
```

Im Ereignis Form_Open des Formulars:

```
Dim nv As CNamedValues

If Len(Nz(Me.OpenArgs,"")) > 0 Then
   Set nv = New CNamedValues
   With nv
      .AsString = Me.OpenArgs
      m_fh.JustHideonOk = .Item("JustHideOnOk")
      m_fh.AutoSave = .Item("AutoSave")
      If .Exists("KundenID") Then
         m_fh.ShowThisID = .Item("KundenID")
      End If
      If .Exists("Name") Then
         strName = .Item("Name")
      Else
         strName = ""
      End If
   End With
   Set nv = Nothing
End If
```

CMoveAndSizeLabel

- Kapselt Funktionalität zur komfortablen **Positionierung** und **Größenänderung** eines Labels (bspw. zur Implementierung einer „Entwurfsansicht zur Laufzeit“ – siehe Berichtseditor)
- Hover-Effekt
- Freie **Konfigurierbarkeit der Darstellung**
- Individuelle **Einschränkungen** bez. minimaler/maximaler Größe, Position
- **Symmetrische Größenänderung**
- Anwendung in **Berichtseditor** für den **Endbenutzer (MDE-fähig**)

CCoolButton

- Kombination aus Bezeichnungsfeld und Befehlsschaltfläche.
- Verhält sich wie eine Befehlsschaltfläche jedoch mit erweiterten **Gestaltungsmöglichkeiten**:
  - Freie Wahl der **Darstellung** (Hintergrund und Rahmen, normal/gedrückt/Fokus)
  - Volle Funktionalität einer Schaltfläche (Fokus, Tastaturkürzel, ...)
  - Erweiterung: **Text und Bild**

```
Private Sub Form_Open(Cancel As Integer)
   Set m_cbOne = New CCoolButton
   Call m_cbOne.SetupControls(Me.cmdOne, _
                              Me.reccmdOne, _
                              Me.lblcmdOne, True)
   Call m_cbOne.SetupFocusAppearance(, , 0, _
                                     RGB(255, 193, 193), 1)
   m_cbOne.PressedBackStyle = bsNormal
   m_cbOne.PressedSpecialEffect = seSunken
End Sub

' Verwendung wie gewöhnliche Befehlsschaltfläche
Private Sub cmdOne_Click()
   MsgBox "You clicked button One", vbInformation
End Sub
```

CFormPosSize

- Stellt **letzte Position und Größe eines Formulars** wieder her
- **Anchoring** (Verankern von Steuerelementen an beliebigen Rändern des Formulars)
- **Minimale/maximale Größe** des Formulars (ohne Bildschirmflackern) mittels Subclassing des Fensters und Abfangen der Windows-Message WM_GETMINMAXINFO

```
Private m_fps As CFormPosSize

Private Sub Form_Open(Cancel As Integer)
   Cancel = Not Right2Open(Me.Name)
   If Cancel Then Exit Sub

   Set m_fps = New CFormPosSize

   Set m_fps.Form = Me
   ' Größenänderung bei Unterformular:
   ' Anker nach allen Seiten
   m_fps.AddAnchor Me.frmsInfos, True, True, True, True
   ' "Kleben" der Schaltfläche an die rechten unteren Ecke,
   ' Anker nach rechts und unten
   m_fps.AddAnchor Me.cmdClose, False, False, True, True
   ' Beschränkung der minimalen Größe des Formulars auf 300x250 Pixel:
   m_fps.MinSize 300, 250
End Sub
```

CKeyDownForwarder

- **Weiterleitung** von KeyDown-Ereignissen **zwischen Formularen**
- **Anwendungsbeispiel**:
  Im gesamten Formular (auch, wenn der Fokus in einem Unterformular ist, kann mit F3 zum Suchsteuerelement gesprungen werden).
- **Kein Code im Quell- (Unter-)Formular**
- Code z.B. im Zielformular:

```
Private Sub Form_Open(Cancel As Integer)
   Set m_kfw = New CKeyDownForwarder
   With m_kfw
      Set .SourceForm = Me.frmsDaten.Form
      Set .DestinationForm = Me
      .ForwardKeyCode = vbKeyF3
   End With
End Sub
```

CDetailsHandler

- Z.B. einfacher Aufruf von Formularen mit entsprechenden Detailinformationen zu einem Wert eines Steuerelements
- Reagiert auf **Tastenkombination** (z.B. Access-konform Strg+F2 – "Zoom")
- Reagiert auf **Mausereignis** (z.B. Doppelklick)
- **Ruft** beliebige **Funktion** auf
- Setzt einheitlichen Steuerelement-**Tipptext**

```
Private Sub Form_Open(Cancel As Integer)
   Set m_dhKunde = New CDetailsHandler
   With m_dhKunde
      Set .EventSource = Me.Kunde
      Set .ValueSource = Me.KundenID
      .DetailExpressionPre = "tdbt_ShowKundenDetails("
      .DetailExpressionPost = ")"
   End With
End Sub
```

CFormAppearance

- Einheitliche Darstellung von **gesperrten Feldern**
- Einheitliche Darstellung von **erforderlichen Feldern**
- Einheitliche Darstellung von ...
- Problemlos erweiterbar

```
   With FormAppearance
      .ResetForNewForm
      .ExcludeLockedControl Me.KursleiterID
      Set .Form = Me
   End With
```

Datengebundene Klassen

Helfer-Klassen mit Datenbankanbindung

CFormHandling

- Funktionalität der **Standard-Schaltflächen** (Abbrechen, Übernehmen, Ok, Neu)
- **Datensatzauswahl** in Einzelformularen durch ein einschränkendes **SQL-Statement** (SELECT ... WHERE ID=...)
- Sicherstellen der **Eingabe erforderlicher Felder** mit definierbaren Meldungstexten, danach Fokus auf das betroffene Feld
- **Protokollierung** von Datensatzneuanlage und –änderung
- **Datensatz-History** (in Zusammenarbeit mit CFormHistories, CFormHistory und CFormHistoryItem)

```
Set m_fh = New CFormHandling
With m_fh
   .NameIDField = "GutscheinID"
   .RecordSource = "SELECT * FROM " & _
                   gc_strTblGutscheine & _
                   " WHERE "
   .FieldPrefix = "Gs_"
   Set .Form = Me
   .RequireControl Me.Gs_Nummer.Name, _
                   "Die Nummer des Gutscheins " & _
                   "muss angegeben werden."
End With
```

CLogger

- **Protokollierung** von Meldungen, Fehlern
- Protokolliert Fehlerort, **geöffnete Formulare und Berichte**
- Stille Protokollierung
- **Beliebig viele Logs** (Fehler, Benutzeraktivitäten, ...)

```
DbLogger.Log -1, _
   "Dieses Formular kann nicht direkt geöffnet werden.", _
   Me.Name, _
   "Form_Open", _
   False, _
   True
```

COptionGlobal, COptionLocal, COptionUser

- Kapselt **Optionseinstellungen**
- **Globale** Optionen: gelten für die **gesamte Anwendung** (z.B. Steuersätze)
- **Lokale** Optionen: gelten für einen **Rechner** (z.B. gemappte Pfade)
- **Benutzer**-Optionen: gelten für einen **Benutzer**, unabh. vom Rechner (z.B. GUI-Einstellungen)
- Ereignis bei Änderung einer Option
- Einfache Implementierung eines ungebundenem Optionen-Dialogs

```
Option("AppBackColor") = RGB(255, 191, 63)
```

Datenkapsel-Klassen

- Strategien zum **Laden**, **Speichern** (Factory-Pattern)
- Einheitliche **Implementierung**
- Starke **Typisierung** durch Eigenschaften für alle Felder der zugrundeliegenden Tabelle
- Auflistungs-Klassen: **Sortierung**, **Filterung** unter Beibehaltung der For Each-Tauglichkeit:

```
Dim objSorter As CQuickSorter
Set objSorter = New CSorter

' entspricht ORDER BY Cust_Year DESC, Cust_LastName ASC
Call objSorter.Sort(Customers, _
   CreateCompCascade( _
      CreateCompInverse(New CCompCustomerYear), _
      New CCompCustomerLastName))
Set objSorter = Nothing

For Each cust in Customers
   ' blabla
Next cust
```

- **Problem**: gebundene Formular umgehen die Idee der Datenkapselung (3-Tier/n-Tier)

Tools zur Klassenentwicklung

Klassenassistenten

Kombinierter Tabellen-/ Klassengenerator

Im Folgenden sind einige Tools aufgeführt, die im Zusammenhang mit der Klassen-Programmierung unter Access hilfreich sein können. Diese Aufstellung erhebt keinen Anspruch auf Vollständigkeit.

Objektbrowser

Der in der VB(A)-IDE integrierte Objektbrowser erlaubt das Untersuchen der Elemente von Klassen. Hier können eingebaute VB(A)-Klassen, Access-Klassen als auch benutzerdefinierte Klassen untersucht werden.
Der Objektbrowser kann mit der Funktionstaste F2 aufgerufen werden.

VB6-Klassenassistent

In der IDE von VB6 ist ein Klassengenerator eingebaut. Der von ihm generierte Code kann relativ problemlos auch als Grundgerüst für Klassen in Access verwendet werden.

Class Builder Wizard

Dieses AddIn von Dev Ashish und Terry Kreft ist ein einfaches Tool zur Erzeugung des Rumpf-Codes einer Klasse. Hierbei ist es jedoch weder möglich, die eingegebenen Daten für die spätere Verwendung oder Änderung zu speichern, noch lassen sich damit in bestehende Klassen weitere Elemente hinzufügen.
Adresse: http://www.mvps.org/access/modules/mdl0023.htm

MZ-Tools

Bekanntes FreeWare COM-AddIn von Carlos J. Quintero. Damit lassen sich unter anderem Vorlagen für Code-Abschnitte definieren und komfortabel einfügen.
Adresse: http://www.mztools.com

softconcept sCodeTools

Wachsende Sammlung von AddIns zur Programmierung in VBA mit besonderer Schwerpunkt auch Access-Entwicklung.

- sClassGen – AddIn zum Erstellen von Klassen und Tabellen
- In Planung: Code-Generator für Datenkapselklassen (Tabellen rein, Klassen raus) gemeinsam mit G. Lesigang
- sCodeDocu – AddIn zur automatischen Erstellung detaillierter HTML-Dokumentation aus dem Quellcode eines Projekts. Beschreibungstexte werden direkt dem Quelltext entnommen, wodurch die konsequente Dokumentation im eigenen Code einen Zusatznutzen erhält.
- scVBDebug – Code-Sammlung zum Debuggen von VBA-Anwendungen. Überprüfung der Aufrufreihenfolge, Ausgabe in XML-Datei (praktisch für die Suche der Ursache eines Access-Absturzes), grafische Aufbereitung im Browser für die spätere Analyse

Adresse: http://www.softconcept.at/sCodeTools

Anhang

Syntax-Auszeichnungen

Die Bedeutung der in diesem Text verwendeten Auszeichnungen der Syntax-Beschreibungen sind wie folgt:

```
<Objektname>
```

Platzhalter
Zu ersetzen durch den konkreten Text.

```
[Optionaler Teil]
```

Optionaler Teil
Kann, muss aber nicht angegeben werden.

```
Alternative1|Alternative2
```

Alternativen
Eine der angegebenen Alternativen muss angegeben werden.

Paarweises Setzen und Löschen von Verweisen

Aufgrund des Referenzzähler-Mechanismus (siehe Abschnitt "Die Lebensdauer eines Objekts" auf Seite 16) ist es notwendig, dass nicht mehr benötigte Verweise auf ein Objekt aufgehoben werden, da andernfalls ein nicht mehr gebrauchter Verweis auf ein Objekt die Freigabe des vom Objekt belegten Speicherplatzes verhindern würde. Obwohl Objektverweise spätestens beim Verlassen des Gültigkeitsbereichs der Objektvariablen durch das Laufzeitsystem automatisch gelöscht werden sollten, ist einem ehebaldigsten expliziten Zerstören der Referenz der Vorzug zu geben.

Aus diesem Grund lege ich dringend folgende SOP (Save Operating Procedure) nahe:

> Schreiben Sie nach einer Zuweisung einer Objektvariablen sofort darauf eine Zeile mit der Zuweisung auf Nothing.
>
> Tun Sie das auch, wenn der Codeabschnitt nur sehr klein und überschaubar sein wird. Wenn Sie dieses Vorgehen immer und ohne Ausnahme machen, werden Sie es auch in unübersichtlicheren Situationen nicht vergessen.

Beispiel:

```
Set frmAktuell = Screen.ActiveForm
Set frmAktuell = Nothing
```

Unmittelbar darauf eingeben!

Fügen Sie *erst jetzt* den Code dazwischen ein:

```
Set frmAktuell = Screen.ActiveForm

' Hier kommt – !!!erst jetzt!!! - Ihr Code

Set frmAktuell = Nothing
```

Besonders bei Objektvariablen wird ein Objektverweis oft in einer Prozedur gesetzt, und in einer anderen erst wieder freigegeben werden. Wenden Sie obige SOP auch auf diese Situation sinngemäß an:

```
Public Sub Init(frm As Access.Form)
   Set mp_frm = frm
End Sub
' ...
Public Sub ShutDown
   Set mp_frm = Nothing
End Sub
```

Unmittelbar darauf eingeben!

Don't: Dim ... As New ...

Vermeiden Sie die folgende Form der kombinierten Deklaration der Objektvariablen und der Instanziierung des Objekts:

```
Dim objChef As New CMitarbeiter
```

Verwenden Sie stattdessen die getrennte Deklaration der Objektvariablen und die Erzeugung einer Referenz:

```
Dim objChef As CMitarbeiter
' ...
Set objChef = New CMitarbeiter
```

Die Verwendung der kombinierten Deklaration und Instanziierung bewirkt, dass das Objekt erst beim ersten Zugriff darauf instanziiert wird:

```
Dim objChef As New CMitarbeiter

objChef.Vorname = "Franz"     ' Erst hier wird objChef instanziiert
```

Dies hat zwei Auswirkungen. Einerseits muss bei jedem Zugriff überprüft werden, ob das Objekt bereits instanziiert wurde, was selbstverständlich zu Lasten der Performance geht.
Zum anderen kann die Überprüfung auf die Zuweisung eines Objektverweises nicht mehr funktionieren:

```
Dim objChef As New CMitarbeiter

If objChef Is Nothing Then  ' Hier wird objChef instanziiert!
   ' ...                   ' Wird nie ausgeführt
End If
```

Standard-Elemente in der VB-Praxis

Die meisten Objekte haben ein Element als Standard-Element definiert (siehe „Standard-Element“ auf Seite 19). Meist ist dies jene Eigenschaft, die die am häufigsten gebrauchte Information des Objekts zurückliefert. Im Falle eines Textfeldes ist es beispielsweise die Eigenschaft `Value`, im Falle eines Bezeichnungsfeldes die Eigenschaft `Caption`.
Die exzessive Verwendung solcher Standard-Elemente hat nun oft unterschätzte und leider auch oft nicht wirklich verstandene Auswirkungen auf die Syntax.

Der Zugriff auf den Wert (Eigenschaft `Value`) eines Steuerelement `txtNachname` müsste vollqualifiziert folgendermaßen aussehen:

```
Me.Controls.Item("txtNachname").Value
```

Wie wir wissen, ist der Aufruf aber eigentlich auch mit der kürzest möglichen Schreibweise

```
txtNachname
```

möglich.

Wie ist diese offensichtliche Verkürzung möglich?

Bei Auflistungen (Collections) ist üblicherweise `Item` die Standard-Eigenschaft, so auch bei der `Controls`-Auflistung eines Formulars oder eines Berichts. Außerdem ist diese `Controls`-Auflistung wiederum das Standard-Element eines Form- oder Report-Objekts. Weiters ist bei einem Textfeld `Value` die Standardeigenschaft und zu guter Letzt können wir im Objektmodul eines Formulars oder Berichts die Angabe von `Me` eigentlich komplett unterlassen, da ohnehin davon ausgegangen wird, dass sich alle Ausdrücke auf das näheste Objekt, nämlich das Formular bzw. den Bericht beziehen. Die Verkürzung ist also durch insgesamt vier Standard-Elemente möglich geworden.

Aufzählungstypen

Seit Access 2000 gibt es die Möglichkeit, benutzerdefinierte Aufzählungstypen zu erstellen. Diese sind zwar kein Klassenspezifikum, in diesem Zusammenhang jedoch oft sehr praktisch. Aufzählungstypen stellen eine Zusammenfassung von Long-Konstanten unter einem Pseudo-Datentyp dar. Dadurch lassen sich zum Beispiel intuitiver nutzbare Fallunterscheidungen vornehmen. Aufzählungstypen werden durch die IDE unterstützt (Intellisense, Autovervollständigen).

Beispiel:

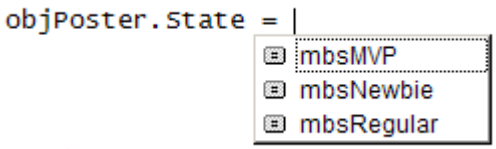
Deklaration:

```
Enum MemberState
   mbsNewbie
   mbsRegular
   mbsMVP
End Enum
```

Verwendung als Pseudo-Datentyp:

```
Public Property Let State(ByVal NewState As MemberState)
   mp_lngState = NewState
End Property
```

Unterstützung durch die IDE:



Standardmäßig werden die Werte der einzelnen Elemente des Aufzählungstyps mit 0 beginnend in der Reihenfolge des Auftretens hochgezählt. Es sind jedoch auch individuelle **Wertzuweisungen** möglich. Die darauf folgenden Elemente erhalten ihre Werte durch fortschreitende Inkrementierung:

```
Enum CustomerErrors
   cerrBase = vbObjectError + 5000
   cerrInvalidID                    ' vbObjectError + 5001
   cerrMissingName                  ' vbObjectError + 5002
End Enum
```

Gerade im Zusammenhang mit Datenbank-Anwendungen können Aufzählungstypen können einen wichtigen Beitrag zur besseren Lesbarkeit von Code führen. So können die ID-Werte von Datensätze einer sogenannten Code-Tabelle (eine Tabelle in der jeder Datensatz zuätzliche Informationen bspw. zu einem Zustand aus wenigen darstellt) in Form eines Aufzählungstyps komfortabel und relativ sicher auch in VB(A)-Code verwendet werden.

Beispiel:

Tabelle tblKursstatus:

tblKursstatus : Tabelle

KursstatusID	Status
1	in Planung
2	anmeldebereit
3	findet statt
4	läuft
5	beendet
6	abgesagt

Aufzählungstyp scKursstati:

```
Public Enum scKursstati
   scKursstatusUnbekannt = 0
   scKursstatusInPlanung = 1
   scKursstatusAnmeldebereit = 2
   scKursstatusFindetStatt = 3
   scKursstatusLaeuft = 4
   scKursstatusBeendet = 5
   scKursstatusAbgesagt = 6
End Enum
```

Benennungs-Konventionen

Folgende Benennungskonventionen in Bezug auf die Members eines Moduls haben sich im Laufe meiner Arbeit mit Klassen herausgebildet und bewährt.:

Präfix	Beschreibung	Zugriff	Beispiel
m_	Interne Member-Variable	Private	m_collItems
mp_	Member-Variable, die Eigenschafts-Wert hält	Private	mp_blnDirty
mpd_	Member-Variable, die Eigenschafts-Werte von Datenbank-basierten Feldern hält	Private	mpd_strLastName
mc_	Konstante	Private	mc_strModuleName
g_	Globale Variable	Public	g_curUStSatz
gc_	Globale Konstante	Public	gc_strAppName

Die Code-Beispiele in diesem Skriptum folgen weitgehend diesen Konventionen.

Glossar

ADH	Access Developer Handbook (siehe "Bücher")
CBF	Code Behind Form Das Objektmodul, das einem Formular (und in ähnlicher Weise einem Bericht und einer Datenzugriffsseite) zugewiesen sein kann.
COM	Component Object Model
Feld	Öffentliche Variable auf Modulebene eines Klassenmoduls. Stellt die einfachste Art der Implementierung einer Eigenschaft dar.
IDE	Integrated Development Environment
RAD	Rapid Application Development
Signatur	Unter der Signatur einer Methode versteht man die Gesamtheit aus der Art der Methode (Funktion oder Prozedur), ggflls. dem Datentyp des Rückgabewerts sowie ggflls. des Aufbaus der Parameterliste. Die formalen Namen sind nicht Teil der Signatur.
UDT	User Defined Datatype (`Type … End Type`)
VBA	Visual Basic for Applications
VBE	Visual Basic Editor

Links

http://www.mvps.org/access/modules/mdl0023.htm	Class Builder Wizard Klassengenerator-AddIn von Dev Ashish und Terry Kreft
http://www.mztools.com	MZ-Tools AddIn für den VBE mit zahlreichen teilweise sehr nützlichen Codierungs-Tools.
http://www-306.ibm.com/software/rational	IBM Rational (vormals Rational Rose) Eine der bekanntesten Data Modeling Suites.
http://smsconsulting.spb.ru/shamil_s/articles.htm	Seite von Shamil Salakhetdinov Einige recht interessante Artikel und Demos zum Thema, DEEP (Dynamic External Event Processing)
http://users.skynet.be/wvdd2/index.html	Seite von Willy Van den Driessche Auf dieser Seite gibt es beinahe Tonnen von Artikel zu fortgeschrittenen Techniken der VB6-Programmierung. Viele Ideen davon lassen sich auch auf VBA umsetzen.
http://www.softconcept.at/tools	Tools zur VBA-Entwicklung mit Schwerpunkt Access (z.B. sClassGen, sCodeGen, sCodeDocu)

Bücher

Litwin, Getz, Gunderloy, Access 2002 Desktop Developer's Handbook, Sybex 2001
ISBN 0-7821-4009-2

Dobson, Programming Microsoft® Access Version 2002 Core Reference, Microsoft Press, 2001
ISBN 0-7356-1405-9

Barker, Microsoft® Access 2000 Power Programming, Sams 1999
ISBN 0-672-31506-8

Lhotka, Visual Basic 6 Business Objects, Wrox Press 1998
ISBN 1-861001-07-X