Reporting

mit Word

und Excel

Appendix:
Und ein wenig Powerpoint

Referent: Michael Zimmermann

Microsoft Most Valuable Professional

Organisation der Datenbankentwickler dbdev.org

Access-Stammtisch Rhein-Main

Zimmermann@szweb.de

Inhalt

Motivation **4**
Ablauf **4**
Anwendungsinstanz **4**
Laufende Instanz nutzen 5
Error 5
API 5
Dokument direkt instanzieren 6
Immer eigene Instanz 7
Sichtbarkeit der Instanz 8
Alternative: Anwendungsloses Arbeiten 9
Dokument erzeugen **9**
Word **10**
Probleme des Makro-Rekorders 10
Objektinstanzierung 10
Methode mit Error oder API 10
Methode mit Dokumentinstanzierung 11
Leeres Dokument, unsichtbar 11
Leeres Dokument, sichtbar 12
Vorhandenes Dokument bearbeiten, unsichtbar 12
Vorhandenes Dokument bearbeiten, sichtbar 12
Vorhandenes Dokument als Vorlage, sichtbar 13
Die wichtigsten Word-Objekte: 13
Das Range-Objekt von Word 15
Die wichtigsten Range-Eigenschaften 16
Verwenden von Textmarken 17
Verwendung von Autotexten 22
Seriendruck per Code 23
Ein paar Grundlagen 30
Excel **32**
Arten von Zugriffsmethoden 32
Objektinstanzierung 33
Methode mit Error oder API 33
Methode mit Dokumentinstanzierung 34
Leere Mappe, unsichtbar 34
Leere Mappe, sichtbar 34
Vorhandene Mappe bearbeiten, unsichtbar 35
Vorhandene Mappe bearbeiten, sichtbar 35
Vorhandene Mappe als Vorlage, sichtbar 35
Die wichtigsten Excel-Objekte 36
Probleme des Makro-Rekorders 38
Worksheets handhaben 38
Das Range-Objekt von Excel 39
Erzeugen von Ranges 39
Range aus Range 42
Arbeiten mit dem Range 42
Schreiben: 45
Lesen: 45
Ausgabe mit Variant-Arrays 47
Formelgruppen 54
Zusammengesetzte Bereiche 56

Informationen über Ranges 56
Adresse eines Ranges 56
Größe eines Ranges 56
Koordinaten eines Ranges 57
Eckzellen als Objekte 57
Bereiche eines zusammengesetzten Ranges: 57
Elternobjekte 57
Gliederungen 58
Formatierungen 59
Schrift 59
Rahmen/Schattierung 59
Textausrichtung 59
Relationale Zugriffe auf Excel 60
CopyFromRecordset 60
SQL 60
DAO-RS 63
ADODB-RS 64
Powerpoint 65
Beispielprozeduren 66
Excel-Beispiele 66
Gruppierter Bericht 66
Periodensystem 68
Auswertung einer poisson-verteilten Probe 73
Daten nach Kategorie nebeneinander 75
CopyFromRecordset-Demo 76

Motivation

- Die Programme sind verbreitet und den Benutzern vertraut
- Anpassungen können daher flexibel auch ohne Programmierkenntnisse vorgenommen werden
- Bei der Seitengestaltung ist man erheblich freier als in einem Report

Ablauf

Das prinzipielle Vorgehen ist bei der Office-Automation immer gleich. Man benötigt einen Objektverweis auf eine laufende Instanz des betreffenden Office-Programms. Dem untergeordnet erzeugt man einen weiteren Objektverweis auf das von dieser Anwendung zu manipulierende Dokument. Die Dokumente erlauben mit verschiedenen Eigenschaften und Bereichsobjekten das Einbringen der Daten aus der Datenbank. Die Ähnlichkeit des Codes der verschiedenen Office-Programme nimmt dabei von der Anwendungsebene bis zur Bereichsebene immer weiter ab.

 Anwendungsinstanz

Zum Erzeugen einer Anwendungsinstanz gibt es mehrere Methoden. Die Office-Programme verhalten sich bereits hier nicht völlig einheitlich.

Man halte sich dabei aber vor Augen, daß das Erzeugen einer Anwendunsinstanz nur Mittel zum Zweck ist. Was wir eigentlich brauchen ist ein Objektverweis auf das zu manipulierende Dokument. Die Anwendung selbst interessiert im wesentlichen nur als notwendiger Container.

Word und **Excel** sind mehrfach instanzierbar, so daß man sich entscheiden muß, ob man laufende Instanzen nutzen will, deren Vorhandensein man dann erst einmal feststellen muß, oder ob man in jedem Fall eine eigene Instanz erzeugt. Beides hat spezifische Vor- und Nachteile.

Outlook und **Powerpoint** lassen nach einer Standardinstallation aufgrund der entsprechenden Registrierungseinträge nur eine laufende Instanz zu, so daß sich hier jedwede weitergehende Überlegung erübrigt.

Laufende Instanz nutzen

Error

Beim Error-Verfahren versucht man einen Verweis auf eine laufende Instanz der Anwendung zu holen. Falls die Anwendung nicht läuft, erzeugt das einen Laufzeitfehler, auf den man reagiert, indem man die Anwendung dann neu instanziert.

```
On Error Resume Next
Set wdApp = GetObject(, "Word.Application")
If wdApp Is Nothing Then _
    Set wdApp = CreateObject("Word.Application")
'oder
If Err.Number <> 0 Then _
    Set wdApp = CreateObject("Word.Application")
On Error GoTo 0
```

```
On Error Resume Next
Set xlApp = GetObject(, "Excel.Application")
If xlApp Is Nothing Then _
    Set xlApp = CreateObject("Excel.Application")
'oder
If Err.Number <> 0 Then _
    Set xlApp = CreateObject("Excel.Application")
On Error GoTo 0
```

API

Das API-Verfahren benutzt – oh Wunder – die entsprechende API-Funktion. FindWindow kann Anwendungen anhand ihres internen Klassennamens oder Fenster aufgrund ihres Titels finden und liefert im Erfolgsfall eine Bezugsnummer zurück, sonst eine 0.

```
Public Declare Function FindWindow _
              Lib "user32" _
              Alias "FindWindowA" ( _
                  ByVal lpClassName As String, _
                  ByVal lpWindowName As String _
              ) As Long
```

```
If FindWindow("OpusApp", vbNullString) = 0 _
    Then
        Set wdApp = CreateObject("Word.Application")
    Else
        Set wdApp = GetObject(, "Word.Application")
End If
```

```
If FindWindow("XLMain", vbNullString) = 0 _
    Then
        Set xlApp = CreateObject("Excel.Application")
    Else
        Set xlApp = GetObject(, "Excel.Application")
End If
```

Die Klassennamen OpusApp und XLMain sind die absurdesten noch nicht; hier ein kleiner Einblick in die finsteren Abgründe schmutziger Programmiererphantasien:

Programm	Klassenname
Access	OMain
Word	OpusApp
Excel	XLMain
PowerPoint95	PP7FrameClass
Powerpoint97	PP97FrameClass
Outlook	rctrl_renwnd32
Rechner	SciCalc
Internet Explorer	IEFrame

Nach diesem Feuerwerk lustiger Einfälle wirkt das Folgende völlig geschmacklos:

Notepad	Notepad

Zum Trost mag gereichen, daß man davon in der Regel allenfalls Word und Excel braucht.

Die so erzeugten Objektverweise auf Anwendungsinstanzen dienen dann dazu, mit ihrer Hilfe Dokumente zu öffnen oder neu anzulegen.

Dokument direkt instanzieren

Die einfachste Methode ist die am wenigsten verbreitete, da man das Schöne und Gute im Gestrüpp des Objektdschungels gern aus den Augen verliert.

Man bekommt, ohne sich über laufende Instanzen Gedanken machen zu müssen, ein neues, leeres Word-Dokument mit dem Einzeiler:

```
Set wdDoc = CreateObject("Word.Document")
```

Man kann ein vorhandenes Word-Dokument öffnen und einen Objektverweis darauf bekommen mit dem Einzeiler:

```
Set wdDoc = GetObject("Pfad\und\Name.doc")
```

Damit hat man die Open-Methode einer laufenden Word-Application simuliert.

Um auf Basis einer Vorlage eine neue Datei zu erzeugen (analog zu Add) braucht es nur eine Zeile mehr:

```
FileCopy "Pfad\Vorlage.doc", "Pfad\Ziel.doc"
Set wdDoc = GetObject("Pfad\Ziel.doc")
```

Das alles ohne Error-Gefummel, API-Gesummse und einem Dutzend Objekten dazwischen.

Nur fürs Protokoll: Damit will ich natürlich nichts gegen so unverzichtbare Techniken wie API und OOP sagen – nein, auf keinen Fall.

Um das abschließend noch einmal klar zu sagen: Die direkte Dokumenterzeugung macht genau das von ganz alleine, was bei anderen Methoden durch viel Code selbst gemacht werden muß: Wenn die Anwendung läuft, wird sie genutzt, wenn nicht, wird sie klaglos gestartet.

Für Excel sieht das Vorgehen so aus:

Leere Mappe:

```
Set xlWbk = CreateObject("Excel.Sheet")
```

Vorhandene Mappe öffnen:

```
Set xlWbk = GetObject("Pfad\und\Name.xls")
```

Vorhandene Mappe als Vorlage:

```
FileCopy "Pfad\Vorlage.xls", "Pfad\Ziel.xls"
Set xlWbk = GetObject("Pfad\Ziel.xls")
```

Der genaue Code für diese Methoden wird bei jedem Programm noch einmal gesondert vollständig vorgestellt, da es jeweils einige kleine Besonderheiten zu beachten gibt, die zwar harmlos sind, aber bei Unkenntnis leicht zu Desparatio developoris suicidalis gallopans (Schnell fortschreitende Programmiererverzweiflung mit Selbstmordgefahr) führen können.

Bei Powerpoint und Outlook hat man zur Nutzung einer laufenden Instanz keine Alternative. Der Code ist entsprechend simpel:

```
Set ppApp = CreateObject("PowerPoint.Application")
```

```
Set olApp = CreateObject("Outlook.Application")
```

CreateObject verhält sich hier von alleine so, wie wir es bei Word und Excel selbst provozieren mußten: Wenn die Anwendung läuft, wird sie genutzt – CreateObject verhält sich in diesem Fall wie GetObject –, wenn nicht wird sie gestartet.

Das Nutzen laufender Instanzen hat entscheidende Vorteile. Es ist insbesondere ressourcenschonend und obendrein, wenn die Instanzen sichtbar sind, was die Regel sein dürfte, für den Benutzer weitaus besser durchschaubar. Mehrere Instanzen können, vor allem, wenn sie sichtbar sind, aber auch bei einem Blick in den Taskmanager, durchaus irritieren.

Immer eigene Instanz

```
Set wdApp = CreateObject("Word.Application")
'oder
Set wdApp = New Word.Application
```

```
Set xlApp = CreateObject("Excel.Application")
'oder
```

```
Set xlApp = New Excel.Application
```

Achtung! Dieser Code hat bei Word und Excel ganz andere Auswirkungen als bei Powerpoint und Outlook!

Es wird so in jedem Fall eine neue Anwendungsinstanz gestartet, auch wenn das Programm bereits läuft. Der Vorteil ist neben dem erheblich einfacheren Code, daß keinerlei Seiteneffekte auftreten können, wenn Code anderer Programme sich ebenfalls an Automation versucht und dabei in der Anwendung womöglich noch mit Selections oder ActiveDocument bzw. ActiveWorkbook arbeitet. Es kann dann ganz schnell passieren, daß Daten irgendwo landen, wo sie keiner haben will.

Der Nachteil ist ganz klar die höhere Speicherbelastung (ca. 20 – 25 MB), bei RAM im GB-Bereich sollte das aber eigentlich kein Thema mehr sein.

Bei Powerpoint und Outlook geht das, wie schon gesagt, sowieso nicht, da hier nur eine Instanz laufen kann.

Sichtbarkeit der Instanz

Alle hier besprochenen Office-Programme unterstützen in ihrem Application-Objekt eine Visible-Eigenschaft, zum Beispiel

```
wdApp.Visible = True 'False
```

Wenn der Benutzer das entstehende Dokument selbst bearbeiten soll, ist klar, daß es sichtbar sein muß. Vollautomatisierte Aufgaben können aber auf Mausklick oder zeitgesteuert unsichtbar Dateien erzeugen, die sie in vorgegebenen Archivordnern ablegen für spätere Verwendung.

Wenn man sich entscheidet, eine unsichtbare Instanz zu verwenden, muß am Ende des Codes auf jeden Fall ein Application.Quit stehen, um die Anwendung zu terminieren, da sie sonst über den Task-manager abgeschossen werden muß. Es muß durch sachgerechtes Error-Handling sichergestellt werden, daß die Terminierung auch im Fehlerfall eintritt.

Die Quit-Methode kann an offenen Dialogen scheitern, die die Programme öffnen, meist Speicherdialoge. Daher sollte man vor dem Quit geöffnete Dokumente speichern oder schließen und die Änderungen verwerfen.

Für Word:

```
'Änderungen speichern
wdApp.Quit wdSaveChanges
'Änderungen verwerfen
wdApp.Quit wdDoNotSaveChanges
'Fragen
wdApp.Quit wdPromptToSaveChanges
```

Für Excel:

```
'Änderungen speichern
For Each xlWbk In xlApp.Workbooks
    xlWbk.Save
Next xlWbk
xlApp.Quit
'Änderungen verwerfen
xlApp.DisplayAlerts = False
xlApp.Quit
xlApp.DisplayAlerts = True
```

Wenn man mit sichtbaren Instanzen arbeitet, sollten diese stehenbleiben, um dem Benutzer nicht vorzugreifen.

Alternative: Anwendungsloses Arbeiten

Word- und Excel-Dokumente können direkt als Dateien geschrieben werden. Hierzu muß das binäre Datei-Format bekannt sein. Das ist zwar miserabel dokumentiert, aber recherchierbar.

Diese recht komplexe Methode verursacht erheblichen Aufwand und würde hier den Rahmen sprengen.

In künftigen Office-Versionen ist das Vorgehen im Blick auf XML eventuell interessanter.

Dokument erzeugen

Die Dokumente, die man mit Informationen aus der Datenbank füllen will, kann man anhand von Vorlagen oder aus leeren Standarddokumenten vollständig per Code erzeugen.

Dieses Erzeugen von Dokumenten nur aus der heißen Luft, die das Kühlgebläse des Rechners zu diesem Zweck zur Verfügung stellt, ist nicht empfehlenswert, da man sich damit sehr viel Codierungsarbeit aufhalst. Wer über eine gerüttelte Portion binären Sportsgeist verfügt, mag aber genau daran seinen Spaß haben.

Für die Normalgebliebenen unter den Entwicklern empfiehlt sich, das grundlegende Design von Hand vorzunehmen – nachdem inzwischen sogar SQL-Statements aus Dutzenden bunter Bildchen zusammengeklickt werden – o tempora, o mores! –, wird man eine Briefvorlage erst recht gemäß dem WYSYWIG-Prinzip nach Augenschein erstellen wollen.

Die Befüllung der so entworfenen Schablone mit Daten nimmt man dann mit Code vor – so hat man unterm Strich am wenigsten Arbeit.

Wenn man sich für die Auslieferung von vorgefertigten Vorlagen mit der Anwendung entschieden hat, kann man diese (langweilig) als Datei mitgeben oder (lustig) als BLOB in der Datenbank.

Wer BLOBs noch nicht kennt, findet im Begleitprogramm ein Beispiel, wie man Dokumente im Binärformat in eine Tabelle speichert und wieder auf die Festplatte schreibt.

Word

Word ist das ideale Instrument für textlastige Berichte (Rechnungen, Lieferscheine, Quartalsberichte), bei denen es auf eine für den Ausdruck optimierte den Schriftsatzregeln entsprechende Präsentation mit vielfältigen Gestaltungsmöglichkeiten ankommt.

Probleme des Makro-Rekorders

Wer sich unter der Ägide desselben schon einmal als Autodidakt betätigt hat, muß jetzt stark sein: Alles vergessen, neu lernen. Der Rekorder arbeitet prinzipbedingt intensiv mit ActiveDocument und Selection, das heißt, mit dem Dokument, das gerade den Fokus hat, und der Auswahl im Text.

Für eine Automation von einem externen Programm aus ist das jedoch völlig ungeeignet. Eine Änderung durch den Benutzer an irgendeinem Dokument der benutzten Instanz kann für den Programmablauf unkalkulierbare Folgen haben. Daher ist mit definierten Objektverweisen und deren Eigenschaften zu arbeiten.

Objektinstanzierung

Zunächst noch einmal der vollständige Code, um eine laufende Word-Instanz mit Verweis auf eine Dokument-Variable zu bekommen.

Methode mit Error oder API

```
Dim wdApp As Word.Application
Dim wdDoc As Word.Document
Dim tDoc As String

tDoc = "Pfad\und\Name.doc"

'Immer eigene Instanz
Set wdApp = CreateObject("Word.Application")

'Sonst
'Methode1:
On Error Resume Next
Set wdApp = GetObject(, "Word.Application")
If Err.Number <> 0 Then _
  Set wdApp = CreateObject("Word.Application")
On Error GoTo 0
```

```
'Methode 1 besser
On Error Resume Next
Set wdApp = GetObject(, "Word.Application")
If Err.Number = 429 _
    Then
       Err.Clear
       Set wdApp = CreateObject("Word.Application")
    Else
        If Err.Number > 0 _
            Then
            MsgBox "Fataler Fehler"
           'Weitere Fehlerbehandlung
        End If
End If
On Error GoTo 0

'Methode2:
If FindWindow("OpusApp", vbNullString) = 0 _
    Then
        Set wdApp = CreateObject("Word.Application")
    Else
        Set wdApp = GetObject(, "Word.Application")
End If
```

Man muß sich natürlich für eine der Methoden entscheiden, also nicht alle in eine Prozedur packen. Es besteht jetzt in jedem Fall ein gültiges wdApp-Objekt. Damit kann man eine Dokumentvariable erzeugen, auf die es eigentlich ankommt:

```
'Neues, leeres Dokument
Set wdDoc = wdApp.Documents.Add
'Neues Dokument von Vorlage
Set wdDoc = wdApp.Documents.Add(tDoc)
'Vorhandenes Dokument öffnen
Set wdDoc = wdApp.Documents.Open(tDoc)
```

Im letzten Fall müßte eigentlich geprüft werden, ob die angeforderte Datei nicht bereits geöffnet ist. Das erübrigt sich aber, da Word in diesem Fall keinen Fehler erzeugt, sondern einfach das betreffende Dokument sang- und klanglos zum aktiven macht.

Damit ist das Ziel erreicht, die gewünschte Datei geöffnet verfügbar zu haben und in wdDoc einen Objektverweis darauf zu halten.

Methode mit Dokumentinstanzierung

Leeres Dokument, unsichtbar

Als erstes ein Beispiel, wie auf diesem Weg ohne Benutzerinteraktion ein Dokument unsichtbar erzeugt, bearbeitet und gespeichert wird:

```
Set wdDoc = CreateObject("Word.Document")
wdDoc.Paragraphs(1).Range.Text = "Hallo, Welt"
wdDoc.SaveAs "Pfad\und\Name.doc"
wdDoc.Close
```

Dieser Code leistet von alleine all das, was man bei den vorangegangenen Methoden erst programmieren mußte. Wenn eine Word-

Instanz besteht, wird sie benutzt, wenn nicht, wird sie automatisch erzeugt. Eine beim Start existierende Instanz bleibt nach dem Schließen des neuen Dokuments stehen, eine eigens eröffnete Instanz wird *ohne weiteres Zutun* automatisch terminiert. Die eigentliche Kernfunktionalität, die bei den Error- und API-Methoden ganze Prozeduren erfordert, steckt hier in der einen einzigen Zeile:

```
Set wdDoc = CreateObject("Word.Document")
```

Leeres Dokument, sichtbar

Das folgende Beispiel macht das erzeugte Dokument statt dessen sichtbar und bietet es zur weiteren Bearbeitung an. Entscheidungen zum Speichern und Schließen des Dokuments oder der Anwendung bleiben dem Benutzer überlassen.

```
Set wdDoc = CreateObject("Word.Document")
Set wdApp = wdDoc.Application
wdDoc.Windows(1).Visible = True '!
wdApp.Visible = True
AppActivate wdApp.Caption
wdDoc.Activate
```

Man beachte, daß nicht nur die Anwendung, sondern auch das Dokument sichtbar gemacht werden muß. Wenn an dem Dokument keinerlei Änderung vorgenommen wird, würde es sonst direkt wieder geschlossen. Visible verhindert das. Der Zugriffscode Windows(1) rührt daher, daß Word es gestattet, vom selben Dokument mehrere Fenster parallel abzuleiten. Standardmäßig gibt es ein Fenster je Dokument.

Vorhandenes Dokument bearbeiten, unsichtbar

Das Öffnen vorhandener Dokumente geht ganz analog. Der einzige Unterschied ist die erste Zeile, die anstelle des Klassennamens den Dateipfad enthält.

```
Set wdDoc = CreateObject("Pfad\und\Name.doc")
wdDoc.Paragraphs(1).Range.Text = "Hallo, Welt"
wdDoc.Save
wdDoc.Close
```

Hier wird Save statt SaveAs *Name* verwendet, da das Dokument ja bereits existiert. Mit SaveAs könnte man allerdings die bearbeitete Datei unter einem neuen Pfad oder Namen speichern und damit das Original erhalten.

Vorhandenes Dokument bearbeiten, sichtbar

```
Set wdDoc = CreateObject("Pfad\und\Name.doc")
Set wdApp = wdDoc.Application
'Hier ggf. Manipulation von wdDoc
wdApp.Visible = True
AppActivate wdApp.Caption
wdDoc.Activate
```

Wenn man eigenen Text aus der Datenbank ins Dokument bringen will, sollte man das tun, bevor man es für den Benutzer sichtbar macht, falls eine Nachbearbeitung erwünscht ist. Danach kann der Benutzer wieder selbst entscheiden, was er mit der angezeigten Datei macht.

Vorhandenes Dokument als Vorlage, sichtbar

Falls anhand einer Vorlage eine neue Datei erzeugt werden soll, die die Vorlage unverändert läßt, besteht bei der obigen Methode das Risiko, daß durch eine Fehlbedienung die Vorlage überschrieben wird. In diesem Fall ist folgende Variante besser:

```
FileCopy "Pfad\Vorlage.doc", "Pfad\NeuDokument.doc",
Set wdDoc = GetObject("Pfad\NeuDokument.doc")
Set wdApp = wdDoc.Application
'Hier ggf. Manipulation von wdDoc
wdApp.Visible = True
AppActivate wdApp.Caption
wdDoc.Activate
```

Für welche Methode man sich auch immer entscheidet, hat man an dieser Stelle nun einen gültigen Verweis auf das gewünschte Dokument. Dieses läßt sich mit vielfältigen Methoden und Eigenschaften manipulieren. Spätestens jetzt wird es also Zeit, sich Gedanken über das Objektmodell von Word zu machen.

Die wichtigsten Word-Objekte:

- Application
  - Documents(i)
    - Paragraphs(i)
    - Tables(i)
      - Cell(i,j)
      - Rows
      - Columns
    - Bookmarks(i)
    - Fields(i)
    - Variables(i)
    - Shapes(i)
    - InlineShapes(i)
      - Aus allen genannten Objekten: **Range**
    - AttachedTemplate
  - MailMerge
  - Templates(i)

- AutoTextEntries(i)
- Styles

Application bezeichnet die Word-Instanz. Viel mehr als die Visible-Eigenschaft [True|False], die die Sichtbarkeit des Programms steuert und die Quit-Methode, die die Instanz beendet, wird man hiervon kaum je brauchen.

Documents ist die Liste der geöffneten Dateien. Die letzte geöffnete Datei hat immer die Bezugsnummer 1. Normalerweise bekommt man einen Dokumentbezug aber nicht über Documents(i), sondern bereits beim Öffnen einer Datei, wie oben gezeigt. Aus diesem leiten wir dann folgende Objekte ab:

Paragraphs sind die Absätze eines Dokuments. Paragraphs(1) der erste Absatz, Paragraphs(2) der zweite und so weiter. Ein Absatz wird in Word an einem ASCII 13 erkannt.

Tables sind die Tabellen eines Dokuments. Man kann Tabellen auch per Code erzeugen durch Tables.Add. Die Steuerung erfolgt über Cell (*ohne* s), Rows und Columns.

Bookmarks sind Textmarken. Textmarken sind eines der wichtigsten Hilfsmittel bei der Programmierung mit Hilfe vorbereiteter Vorlagen. Es gibt zwei Typen, die I-Bookmark und die [ ]-Bookmark, die sich beim Beschreiben unterschiedlich verhalten.

Fields sind die Feldfunktionen eines Dokuments. Bei der Automation von einer Datenbank aus sind sie meist weniger interessant.

Variables sind sogenannte Dokumentvariablen. Damit kann man Informationen unsichtbar ablegen, die dann nur per Code oder Feldfunktion auslesbar sind.

Shapes sind alle graphischen Obekte, die frei verschiebbar sind, also Bilder, Textfelder etc. Man kann vorhandene ansprechen sowie neue erzeugen.

InlineShapes sind Graphiken, die den Textfluß „Mit Text in Zeile“ aufweisen.

Range ist ein Bereich. Aus allen oben genannten Objekten läßt sich – meist über eine Range-Eigenschaft – ein Range-Objekt erzeugen, das für den Bereich, den das Objekt einnimmt, steht und über das das Mutterobjekt manipuliert wird. *Das Range-Objekt ist der Schlüssel zur Word-Programmierung.*

MailMerge repräsentiert die Seriendruckfunktionalität von Word. Das ist auch bei Automation interessant, da man so beispielsweise aus einer gewöhnlichen Briefvorlage per Code eine Seriendruck-vorlage erzeugen kann.

Templates sind Word-Vorlagen (dot-Dateien). Aus historischen Gründen geht bei Word die Bedeutung einer Template aber weit über eine schlichte Dateivorlage hinaus. Es muß immer wenigstens

eine Vorlage mit Word geladen werden (in der Regel die Normal.dot) und jedes Dokument ist mit einer Vorlage verknüpft (in der Regel wieder die Normal.dot), die die verfügbaren AutoTexte und Formatvorlagen enthält.

AttachedTemplate ist die eben angesprochene verknüpfte Vorlage eines Dokuments.

AutoTextEntries(i) sind die in einer Vorlage verfügbaren AutoTexte.

Styles(i) sind die in einer Vorlage bzw. einem Dokument verfügbaren Formatvorlagen.

Das Range-Objekt von Word

Über die Zuweisung des Dokuments an eine Objektvariable und das Arbeiten mit dem Range-Objekt wird Zugriffssicherheit erzielt, die unabhängig von Selektionen und obendrein schneller ist.

Achtung! Es gibt ein Range-Objekt, eine Range-Methode und eine Range-Eigenschaft!

Die Auswahl (Selection) deckt sich zum großen Teil in Methoden und Eigenschaften mit Range, ist aber ein eigenes Objekt vom Typ Selection und hat eine Range-Eigenschaft. Das Objektmodell ist an dieser Stelle insoweit etwas unlogisch.

Um die Eigenschaften eines Range-Objekts nutzen zu können, muß man zunächst eines erzeugen.

```
Dim wdRng As Word.Range
```

Das Dokument selbst hat eine Range-*Methode* mit den optionalen Parametern Start, End und ergibt als Rückgabe ein Range-Objekt.

Ohne Angabe von Parametern umfaßt der Range das gesamte Dokument, mit Parametern den Bereich zwischen den angegebenen Zeichenpositionen.

Ganzes Dokument

```
Set wdRng = wdDoc.Range()
```

Zeichen 3 bis Zeichen 12

```
Set wdRng = wdDoc.Range(3, 12)
```

Praktisch ist diese Art, ein Bereichsobjekt zu erzeugen, kaum von Bedeutung.

Weitaus interessanter ist das Erzeugen von Ranges aus bereits existierenden Objekten. Grundregel: Viele anschauliche Word-Objekte (Absatz, Tabelle, Autoform etc.) haben eine Range-Eigenschaft, mit der auf sie zugegriffen werden kann.

Range-Objekte werden fast immer aus einer Range-Eigenschaft eines Objektes erzeugt.

```
Set wdRng = wdDoc.Paragraphs(2).Range
'Zweiter Absatz
Set wdRng = wdDoc.Tables(3).Range
'Gesamte dritte Tabelle
Set wdRng = wdDoc.Tables(3).Rows(1).Range
'Erste Zeile der dritten Tabelle
Set wdRng = wdDoc.Tables(3).Cell(3, 5).Range
'Zelle 3, 5 der dritten Tabelle
Set wdRng = wdDoc.Bookmarks(4).Range
'Vierte Textmarke
Set wdRng = wdDoc.Bookmarks("Textmarkenname").Range
'Textmarke mit dem Namen "Textmarkenname"
Set wdRng = wdDoc.Fields(4).Range
'Viertes Feld
```

Die mit Abstand wichtigste Methode ist dabei das Erzeugen eines Bereichs aus einer Textmarke. Von allen obengenannten Objekten ist die Textmarke nämlich als einzige nicht nur über einen numerischen Index sondern über einen Namen referenzierbar.

Das ist geradezu unverzichtbar, da man beim Entwickeln die numerischen Indizes, die Objekte später in einem auszugebenden Dokument haben werden, im Zweifelsfalle nicht kennt. Um nun auch Tabellen oder Feldern Namen geben zu können, ist es ratsam, diese in Textmarken einzufassen. Über diesen kleinen Umweg kann man dann zugriffssicher auch diese Elemente ansprechen.

Die Textmarke (Bookmark) erweist sich so als das überragend wichtige Objekt, um einen Range zu erzeugen. Daher wird ihr auch das gesamte übernächste Kapitel gewidmet.

Die wichtigsten Range-Eigenschaften

Die in der Praxis interessantesten Eigenschaften eines Ranges sind in der folgenden Hierarchie dargestellt.

- Range
  - .Text
  - .Font
    - .Bold
    - .Italic
    - .Name
    - .Size
  - .Border
  - .Shading

Text legt – oh Wunder – den Text eines Ranges fest. Für das Beschriften vorhandener Vorlagen die weitaus relevanteste Eigenschaft.

Font steht für das Schrift-Objekt eines Bereichs. Mit dessen Eigenschaften

- **Bold** – Fett; True oder False
- **Italic** – Kursiv True oder False
- **Name** – Schriftart; String
- **Size** – Schriftgröße in Punkt, Long

kann man die Schrift formatieren.

Ebenfalls spannend sind **Border** – Rahmen – und **Shading** – Schattierung –, mit denen man Absätze oder Tabellen mit Rahmen und Hintergründen versehen kann.

Verwenden von Textmarken

Die Textmarke ist der klare Königsweg, um ein Dokument als Vorlage für Automation vorzubereiten. Wie schon eingangs erwähnt, gibt es zwei Typen von Textmarken: die I-Bookmark und die [ ]-Bookmark

Man fügt Textmarken mittels *Einfügen* > *Textmarke*…in ein Dokument ein. Welcher Textmarkentyp dabei entsteht, hängt von der dabei gesetzten Markierung ab.

Wenn eine blinkende Einfügemarke steht (keine Markierung) wird eine I-Bookmark erzeugt, wenn eine Bereichsmarkierung besteht, wird eine [ ]-Bookmark erzeugt, die die gesetzte Markierung umschließt.

Da man im allgemeinen bei der Automation aus einer Datenbank heraus Dokumente auf Basis einer immer wieder verwendeten Vorlage erzeugen wird, ist die einfacher zu programmierende [ ]-Bookmark vorzuziehen. Am anschaulichsten ist es, einen Platzhaltertext zu verwenden, der dem Namen der Textmarke entspricht – so hat man den besten Überblick. Ein Beispiel:

Absender (konstant)

ANSCHRIFT

DATUM

BETREFF

TEXT

Wenn man über *Extras* > *Optionen* > *Ansicht* > *Textmarken* dieselben anzeigen läßt, sollte obiges Beispiel folgendes Bild geben:

Absender (konstant)

[ANSCHRIFT]

[DATUM]

[BETREFF]

[TEXT]

Beim Ausführen des Programmcodes werden diese Platzhaltertexte dann durch die echten Werte überschrieben, wobei die Textmarke gelöscht wird. Diese Vernichtung der [ ]-Textmarke beim Beschreiben ist nicht störend, da die Vorlage, von der man sein aktuelles Dokument zieht, ja hierbei nicht verändert wird.

Um eine Tabelle oder ein Feld mit Hilfe einer Marke anzusprechen, würde man einfach die gesamte Tabelle oder das Feld markieren und dann genau wie sonst auch *Einfügen* > *Textmarke*.

Im folgenden Beispielcode gehe ich davon aus, daß eine Variable wdDoc, wie eingangs gezeigt, mit einem Verweis auf das zu bearbeitende Dokument existiert, von dem wir voraussetzen, daß es die Textmarken aus den Beispiel oben enthält, sowie eine Tabelle, die in eine Textmarke namens TABELLE eingefaßt ist; ebenso ein Feld, das in eine Textmarke DASFELD eingefaßt ist.

```
Dim wdRng As Word.Range
Dim wdTab As Word.Table
Dim wdFld As Word.Field

Set wdRng = wdDoc.Bookmarks("Anschrift").Range
wdRng.Text = "Herrn" & Chr(11) & _
             "Bernd Beispiel") & Chr(11) & _
             "Beispielweg 7") & Chr(11) & Chr(11) & _
             "12345 Beispielbach
Set wdrng = wdDoc.Bookmarks("DATUM").Range
wdRng.Text = Format$(Date(), "dd.MM.yyyy")
Set wdrng = wdDoc.Bookmarks("BETREFF").Range
wdRng.Text = "Rechnung"
Set wdrng = wdDoc.Bookmarks("TEXT").Range
wdRng.Text = "Bitte zahlen Sie schnell und viel."
```

Das Beispieldokument sähe jetzt so aus:

Absender (konstant)

Herrn
Bernd Beispiel
Beispielweg 7

12345 Beispielbach

03.07.1789

Rechnung

Bitte zahlen Sie schnell und viel.

Im wahren Leben werden die Textliterale natürlich mittels Recordsets aus der Datenbank geholt.

```
'***
Set wdTab = wdDoc.Bookmarks("TABELLE").Range.Tables(1)
Set wdFld = wdDoc.Bookmarks("DASFELD").Range.Fields(1)
'***
```

Folgenden Dingen gebührt hier erhöhte Aufmerksamkeit:

- Für den Zeilenwechsel in der Anschrift ist ein ASCII 11 zu verwenden und nicht etwa ASCII 13 oder vbCrLf. Ein ASCII 13 bedeutet in Word einen Absatzwechsel und das ist etwas ganz anderes als ein erzwungener Zeilenwechsel innerhalb eines Absatzes.
- Der Trick bei der Tabellenreferenzierung beruht darauf, daß der Range der Textmarke TABELLE, nur genau eine Tabelle umfaßt. Mit der Tables-Auflistung des Ranges bekommt man alle Tabellen innerhalb des Bereichs. Da es nur eine gibt, hat diese sicher die relative Bezugsnummer Tables(1), unabhängig davon, ob sie im Dokument nun auch Tables(1) oder halt Tables(7) oder was auch immer ist.
- Entsprechendes gilt genauso für das Feld.

Weiter im Code:

```
Set wdRng = wdTab.Cell(1,1).Range
wdRng.Text = "Überschrift 1"
'Auch mal ohne Range-Variable:
wdTab.Cell(1,2).Range.Text = "Überschrift 2"
wdTab.Cell(1,3).Range.Text = "Überschrift 3"

wdTab.Cell(2,1).Range.Text = "Wert 1,1"
wdTab.Cell(2,2).Range.Text = "Wert 1,2"
wdTab.Cell(2,3).Range.Text = "Wert 1,3"
```

In der Praxis würde man eine solche Tabelle – zum Beispiel für Rechnungspositionen – in der Vorlage so vorbereiten, daß man sie mit der erforderlichen Spaltenanzahl und *genau zwei Zeilen* ausstattet. Die erste Zeile enthält die fixen Überschriften, die zweite Zeile ist leer, aber mit den gewünschten Formaten versehen (Schriftart, -größe, rechts-, linksbündig etc.).

Bei der Ausgabe per Code werden die nötigen Datenzeilen dann dynamisch erzeugt:

```
'Angenommen, Recordset und Tabelle haben drei
'Spalten auszugeben
n = rcsPositionen.RecordCount
i = 1
With rcsPositionen
    Do Until .EOF
        i = i + 1
        wdTab.Cell(i, 1).Range.Text = "" & rcs.Fields(0).Value
        wdTab.Cell(i, 2).Range.Text = "" & rcs.Fields(1).Value
        wdTab.Cell(i, 3).Range.Text = "" & rcs.Fields(2).Value
        wdTab.Rows.Add
        .MoveNext
    Loop
End With
Set wdRng = wdTab.Rows(1).Range
wdRng.Borders(wdBorderBottom).LineStyle = wdLineStyleDouble
```

Das Schöne daran ist, daß Rows.Add alle Formate der Vorgängerzeile komplett übernimmt. Das ist auch der Grund für die vorformatierte leere Datenzeile. Den Fettdruck beispielsweise der ersten Überschriftenzeile will man ja nicht nach unten übernehmen.

Natürlich kann man, wenn man seine Vorlagen stärker generalisieren will, auch die Überschriften der Tabelle dynamisch befüllen (wdTab.Cell(1, j).Range.Text) und auch Spalten dynamisch anfügen (wdTab.Columns.Add).

Auch ein Erzeugen von Tabellen aus der hohlen Hand an einer vorgegebenen Position ist problemlos möglich. Angenommen, man möchte bei einer Textmarke TABELLEHIERHIN eine Tabelle mit fünf Spalten und zwölf Zeilen erzeugen:

```
Set wdRng = wdDoc.Bookmarks("TABELLEHIERHIN").Range
Set wdTab = wdDoc.Tables.Add (wdRng, 12, 3)
```

Das wdTab-Objekt kann wieder beliebig bearbeitet werden.

Wir sind bisher immer davon ausgegangen, daß die Textmarke, die zur Zieldefinition irgendwelchen Codes benutzt wurde, vom Typ [ ] war.

Der Vorgabetext einer solchen Marke wird beim Beschreiben ersetzt und die Marke zerstört, so daß sie nicht ein zweites Mal als Anker dienen kann. Falls es aus irgendwelchen Gründen erforderlich sein sollte, die Textmarke zu erhalten, kann man das mit folgendem Trick erreichen:

```
Set wdRng = wdDoc.Bookmarks("TEST").Range
wdRng.Text = "So ein schöner Text."
```

```
wdDoc.Bookmarks.Add "TEST", wdRng
```

Der Bereich der Textmarke wird einer Range-Variablen zugewiesen, beim folgenden Schreibvorgang wird die Marke zerstört, aber dann direkt mit der in wdRng gespeicherten Bereichsinformation mit ihrem ursprünglichen Namen wieder neu erzeugt.

Das nächste Beispiel zeigt, wann und wie die bisher stiefmütterlich behandelte I-Textmarke sinnvoll ist. Zunächst bleibt eine solche Marke beim Beschreiben per se erhalten, da ein Text *hinter* ihre virtuelle Zeigerposition geschrieben wird. Das hat zur Folge, daß bei mehrmaliger Verwendung die Textblöcke in umgekehrter Reihenfolge erscheinen:

```
wdDoc.Bookmarks("ITEST").Range.Text = "Erstens"
```

Im Dokument:

I Erstens

```
wdDoc.Bookmarks("ITEST").Range.Text = "Zweitens"
```

Im Dokument:

I ZweitensErstens

```
wdDoc.Bookmarks("ITEST").Range.Text = "Drittens"
```

Im Dokument:

I DrittensZweitensErstens

Das Verhalten ist nützlich, um auf einfache Weise z. B. bei einer Adresse mit nur einer Textmarke ADRESSE die PLZ-Ort-Zeile fett zu machen:

```
Set wdRng = wdDoc.Bookmarks("ADRESSE").Range
wdRng.Text = "12345 Meierbach"
wdRng.Font.Bold = True
'Erst den Ort ...
Set wdRng = wdDoc.Bookmarks("ADRESSE").Range
wdRng.Text = _
    "Herrn" & Chr(11) & _
    "Peter Meier" & Chr(11) & _
    "Meiersweg 12" & Chr(11) & _
    Chr(11)
wdRng.Font.Bold = False
'... dann Name und Straße
```

Der gewünschte Effekt wird hier erzielt durch die zweimalige explizite Zuweisung des Ranges an eine Variable.

Ergebnis:

I Herrn
Peter Meier
Meiersweg 12

12345 Meierbach

Für vorsichtige Naturen gibt es noch die Möglichkeit, zu prüfen, ob eine bestimmte Textmarke im Dokument überhaupt vorhanden ist:

```
If wdDoc.Bookmarks.Exists("ADRESSE") Then ...
```

Das ist übrigens kein Vertipper, die Syntax ist wirklich so komisch. Wer selbst häufig Klassen erstellt, hätte wohl eher etwas wie

```
wdDoc.Bookmarks("ADRESSE").Exists
```

implementiert, aber es ist halt, wie es ist.

Verwendung von Autotexten

Wenn zum Beispiel in Geschäftsbriefen immer wieder gleiche Floskeln eingesetzt werden sollen, könnte man solche Texte natürlich in der Datenbank ablegen. Wer die 1. Normalform ernstnimmt, wird aber bereits mißtrauisch, wenn ein Feldeintrag deutlich mehr als etwa ein Dutzend Zeichen hat. Und viele Word-Benutzer haben all diese Texte in Word sowieso schon als AutoTexte abgelegt. Könnte man da nicht…? Man kann.

Wie schon weiter oben erwähnt, sind Textbausteine in Dokument-vorlagen (Templates, .dot) abgelegt. Jeder dieser Textbausteine muß innerhalb einer Vorlage einen eindeutigen Namen haben. Vorgabe sind die ersten Zeichen des Eintrags; gewiefte Word-Nutzer vergeben aber selbstverständlich eigene sprechende Namen.

Der Zugriff auf vorhandene AutoTexte ist denkbar einfach:

```
Dim wdDot As Word.Template

Set wdRng = wdDoc.Bookmarks("TEST").Range
Set wdDot = wdDoc.AttachedTemplate
'für mit dem Dokument verbundene Vorlage
Set wdDot = wdApp.NormalTemplate
'für Standard-Vorlage Normal.dot

wdDot.AutoTextEntries("Test").Insert wdRng
```

wdRng ist hier ein Parameter der Insert-Methode. Man kann hier einen Range natürlich auch direkt, also ohne Zuweisung an eine Range-Variable angeben. Daß der Autotext und die Textmarke den gleichen Namen haben, ist nicht zwingend, aber deutlich über-sichtlicher. Grundsätzlich ist man in der Wahl beider Namen aber frei.

Falls man selbst Autotexte in Word erstellen will, ist folgendes zu beachten:

Es gibt zwei Arten von Autotexten, die sich unterschiedlich verhalten, aber davon abgesehen nicht unterscheiden lassen:

Wenn beim Erstellen eines Autotextes die Textmarkierung die letzte Absatzmarke **nicht** umfaßt, entsteht ein Autotext vom Typ Text, wenn beim Erstellen die letzte Absatzmarke **mitmarkiert** ist, entsteht ein Autotext vom Typ Absatz. Absatzmarken innerhalb des markierten Textteils sind dabei belanglos; es kommt nur auf die letzte an.

Ein Autotext vom Typ Text paßt sich beim Einfügen in der Formatierung (Schriftart, -größe, Bündigkeit etc.) seiner Umgebung an wird also als Plain Text eingefügt.

Ein Autotext vom Typ Absatz enthält alle Formatierungen, die bei seiner Erstellung aktiv waren, mitgespeichert. Er wird also mit seiner eigenen Schriftart, Ausrichtung etc. eingefügt, unabhängig davon, wie seine Umgebung formatiert ist.

Möglich ist natürlich auch, daß man einen halbfertigen Brief mit Adresse dem Benutzer präsentiert und es diesem überläßt, innerhalb der Word-Oberfläche das Schreiben mit seinen Autotexten händisch zu komplettieren.

Seriendruck per Code

Wann braucht man überhaupt Seriendruck? Als DB-Entwickler mit profunden Automationskenntnissen eigentlich höchst selten. Seriendruck ist dann unverzichtbar, wenn ein User, basierend auf einer Datenzusammenstellung, eine Dokumentschablone selbst frei entwerfen können soll, wobei er die Seriendruckfelder als Platzhalter für die jeweiligen Daten einfügt. Das Ergebnis ist eine große Datei, die je Datensatz einen Abschnittswechsel enthält (neuer Empfänger, neue Seite in einem Brief zum Beispiel) und nach dem Ausdruck im allgemeinen nicht archiviert wird, da sie viel Platz belegt und eigentlich keine relevanten Informationen enthält: Die veränderlichen Daten (Kundenadressen o. ä.) stehen eh' in der Datenbank, der unveränderliche Text ist tausendmal derselbe.

Wenn man Dokumente voll automatisiert ausgeben kann, kann man dasselbe mit einzelnen Dateien, die man genauso verwirft wie die eine große, mit einer schlichten Schleife im Code auch haben – ohne Seriendruck.

Der häufigste Grund, warum in Betrieben Seriendruck – sogar für einzelne(!) Briefe verwendet wird, ist meiner Erfahrung nach schlicht der, daß dort einfach keiner weiß, wie man Daten sonst von Access oder Excel nach Word bekommt.

Wenn man denn den Seriendruck von Word braucht, kann man aber natürlich auch hier mit Hilfe von Code alles Automatisierbare vorbereiten. Es ist grundsätzlich nicht empfehlenswert, fertige Vorlagen mit Seriendruckinformationen anzulegen. Man kann recht problemlos ein Standard-Word-Dokument (also ohne Seriendruckeigenschaft) mittels Code bei Bedarf zu einem Seriendruckhauptdokument machen.

Die Vorteile liegen auf der Hand: Man pflegt zum Beispiel nur eine Vorlage Brief.doc, die ja unter Umständen an ein verändertes Corporate Design anzupassen ist, statt, wie schon gesehen, eine ganze Herde davon, die da heißen Einzelbrief.doc,

SerienbriefMitglieder.doc, SerienbriefKunden.doc und so weiter. Redundanz ist nicht nur bei den Nutzdaten von Übel.

Obendrein hat man nicht das Problem, daß die Vorlage nicht mehr funktioniert, weil sich der Name oder der Speicherort der Datenquelle geändert hat. Wenn man die Seriendruckfunktion jeweils per Code erzeugt, sind alle Pfade immer aktuell.

Das grundsätzliche Vorgehen besteht darin, daß man einen passenden Connection-String zusammenbastelt – in so etwas sind Datenbänker ja recht erfahren –, dem Dokument sagt, es sei ein Seriendruckdokument und dann an den passenden Stellen, die man im Zweifel durch Textmarken identifiziert, Seriendruckfelder einfügt, die zu den Feldern der Datenquelle passen.

Man kann hierbei verschiedene Typen von Seriendokumenten erzeugen, von denen eigentlich nur der Serien**brief** regelmäßig nützlich ist. Gelegentlich mag man auch einmal einen Serien**katalog** oder **Etiketten** gebrauchen können. Der Brief trennt die aus literalem Text und Feldfunktionen gebildeten Blöcke durch seitenweise Abschnittswechsel, der Katalog führt fortlaufende Abschnitte ein, Etiketten werden auch nebeneinander angeordnet.

Es gibt drei Schnittstellen, über die der Datenaustausch stattfinden kann: DDE, ODBC, OleDB. Welche Schnittstelle angesprochen wird, erkennt Word am Connection-String. Dabei sollte folgendes beachtet werden:

In alten Word-Versionen (bis einschließlich 2000) ist DDE Standard und ODBC als Alternative möglich. Ab Word XP ist das neue OleDB verfügbar und dort auch Standard. Ein SD-Dokument, das mit OleDB erstellt wurde, zum Beispiel händisch mit XP, wird von älteren Word-Versionen überhaupt nicht als gültiges Seriendruck-dokument erkannt.

Die DDE-Schnittstelle sollte man gar nicht verwenden. DDE ist veraltet, langsam und problembehaftet. DDE öffnet die Datenquelle beim Aufruf erneut, auch wenn sie schon offen ist, was bei Access als Quelle zu Sperrkonflikten führen kann.

Die ODBC-Schnittstelle ist brauchbar und daher bei Word bis 2000 das Mittel der Wahl.

Die ab XP verfügbare OleDB-Schittstelle ist ebenfalls recht zuverlässig. Sie ist stabil und problemlos, allerdings etwas langsamer als ODBC. Bei sehr umfangreichen Seriendrucken kann letzteres also noch als Alternative zu OleDB gelten.

Kurz und gut: Wenn der Code auch mit alten Word-Versionen laufen muß oder Performance-Probleme auftreten, ODBC verwenden, sonst OleDB.

Beim ODBC-Seriendruck gibt es leider auch einige Schattenseiten. Abhängig vom Datentyp und den Ländereinstellungen kann in Word gerne bei Währungen oder Zeitdaten rechter Unfug ankommen.

Ich habe mir daher schon früh angewöhnt, als Quelle für einen Seriendruck immer eine temporäre Tabelle zu benutzen, die ausschließlich Textfelder hat, und Datumsangaben, Beträge und ähnliches dort als formatierten String abzulegen. Word ist nun mal eine **Text**verarbeitung und kommt daher auch am besten mit Texten klar. Außerdem umgeht man damit von vornherein userverwirrende Eingabefenster, die nach irgendwelchen Parametern fragen…

Mit diesem Vorgehen ist man auch bei OleDB in jedem Fall auf der sicheren Seite.

Noch zwei Macken seien erwähnt:

Man muß innerhalb der Seriendruckfunktionalität einen SQL-String zusammenbauen, der von Word ausgeführt wird. Bei dem eben vorgeschlagenen Vorgehen wird dieser regelmäßig einfach nur lauten:

```
SELECT * FROM [tblQuelle]
```

Man achte auf die eckigen Klammern. Das ANSI-konforme

```
SELECT * FROM tblQuelle
```

wird nicht ausgeführt!

Wer einmal Makrorekorder-Code gesehen hat, dem ist vielleicht folgende Syntax aufgefallen:

```
SELECT * FROM `tblQuelle`
```

Die schrägen Hochkommata (ANSI 96) meint Word bitterernst. Wer hier nichts oder ein einfaches Hochkomma (') verwendet, wird mit nichtssagenden Fehlermeldungen nicht unter drei Stück bestraft. Die einzige ästhetisch befriedigende Alternative zu diesem Fliegenschiß sind die eckigen Klammern.

Wenn man mit der Execute-Methode eine Ausgabe aller Briefe in ein neues Dokument vornimmt, wird man auf dieses Dokument auch im Code zugreifen wollen. Jeder Klassenprogrammierer hätte daher Execute als Function implementiert, die einen Verweis auf das neue Dokument zurückgibt. Nicht so die Word-Programmierer! Die haben tatsächlich eine Sub verwendet, so daß das neue Dokument haltlos im RAM herumgeistert. Da Word dem neuesten erzeugten Dokument immer den Index eins zuschiebt, gibt es aber Abhilfe. Wenn man direkt nach Execute einen Verweis auf Documents(1) holt, stehen die Chancen, damit das neue Dokument erwischt zu haben, aber immerhin so gut, daß ich damit in über 15 Jahren nie einen Fehler bekam.

Schauen wir uns jetzt den Code dazu an:

```
Dim wdApp As Word.Application
Dim wdDoc As Word.Document
Dim wdDocSDAusgabe As Word.Document
Dim wdRng As Word.Range
Dim wdMerge As Word.MailMerge
Dim wdMfd As Word.MailMergeField
Dim wdFld As Word.Field
```

```
Dim tConnectionString As String
Dim tSourceDatabase As String
Dim tSourceTable As String

Set wdDoc = <Verweis auf Briefvorlage nach irgendeiner
             der eingangs erläuterten Methoden>
Set wdMerge = wdDoc.MailMerge
```

Mit dieser Zeile ist im Grunde noch nichts passiert. wdMerge enthält jetzt einen Verweis auf die hypothetische Seriendruck-funktionalität von wdDoc, ohne daß dieses jetzt bereits ein Seriendruckdokument wäre.

```
tSourceDatabase = "Pfad\und\Name.mdb"
                  'oder auch CurrentDb.Name
tSourceTable = "tblSD" 'Tabellen- oder Abfragename
```

Die Namen sind passend zu vergeben.

Nachfolgend der Code für OleDB:

```
'Oledb
tConnectionString = _
    "Provider=Microsoft.Jet.OLEDB.4.0;" & _
    "Password="""";" & _
    "User ID=Admin;" & _
    "Data Source=" & tSourceDatabase & ";" & _
    "Mode=Read;" & _
    "Extended Properties="""";"
```

Der Connection-String ist gegebenenfalls anzupassen, zum Beipiel der Provider, wenn auf andere DB-Formate zugegriffen werden soll. Die Syntax erinnert stark an ADO-Connection-Strings.

```
wdMerge.OpenDataSource _
    Name:=tSourceDatabase, _
    LinkToSource:=True, _
    Connection:=tConnectionString, _
    SQLStatement:= _
           "SELECT * FROM [" & tSourceTable & "]", _
    SQLStatement1:="", _
    SubType:=wdMergeSubTypeAccess
```

Hierdurch wird das Dokument zum Seriendruckdokument und ist mit der angegebenen Quelle verknüpft. Man kann der guten Ordnung halber noch explizit den Seriendrucktyp angeben. Wenn man darauf verzichtet, wird als Standard wdFormLetters verwendet.

```
wdMerge.MainDocumentType = wdFormLetters
```

Andere interessante Typen sind:

- wdCatalog
- wdMailingLabels
- wdNotAMergeDocument

Damit kann man Kataloge (ohne Seitenwechsel) und Etiketten drucken. Mit wdNotAMergeDocument wird die Seriendruck-eigenschaft aufgehoben.

Hier noch das gleiche für ODBC/DDE:

```
'ODBC
tConnectionString = _
    "DSN=Microsoft Access-Datenbank;" & _
    "DBQ=" & tSourceDatabase & ";" & _
    "DriverId=25;" & _
    "FIL=MS Access;" & _
    "MaxBufferSize=2048;" & _
    "PageTimeout=5;"

'DDE
tConnectionString = "TABLE " & tSourceTable

'ODBC/DDE
wdMerge.OpenDataSource _
    Name:=tSourceDatabase, _
    LinkToSource:=True, _
    Connection:=tConnectionString, _
    SQLStatement:= _
           "SELECT * FROM [" & tSourceTable & "]", _
    SQLStatement1:=""
```

Falls sich jemand fragt, was es mit SQLStatement1 auf sich hat: Die Eigenschaft SQLStatement kann nur 255 Zeichen aufnehmen. Mit SQLStatement1 kann man um weitere 255 Zeichen aufstocken. Da man, wie schon erwähnt, ein komplexes SQL nicht in Word aufbaut, braucht das aber kein Mensch. Der Hintergrund sind die Sortier- und Filterungsoptionen im Seriendruck von Word, die aber bei Automation aus einer DB-Anwendung heraus überflüssig sind.

Nachfolgend werden die Seriendruckfelder ins Dokument eingefügt. Die Namen der Serienduckfelder sind dabei *nicht* frei wählbar, sondern sie müssen mit denen der Felder der Tabelle oder Abfrage übereinstimmen, deren Inhalte sie darstellen sollen.

Vorab sollte man einen Range definiert haben, der das Feld aufnimmt:

```
Set wdRng = wdDoc.Bookmarks("Anschrift").Range
```

Für das Einfügen der Felder gibt es etliche Syntax-Varianten, die sich in der Leistung nicht unterscheiden. Man ist hier also allein auf seinen Geschmack angewiesen.

```
Set wdMfd = wdMerge.Fields.Add(wdRng, "Anschrift")

Set wdFld = wdRng.Fields.Add( _
              wdRng, wdFieldMergeField, "Anschrift")

Set wdFld = wdDoc.Fields.Add( _
              wdRng, wdFieldMergeField, "Anschrift")
```

Da man den sich ergebenden Objektverweis auf das eingefügte Feld in aller Regel nicht benötigt (das gilt so nur für Seriendruckfelder; bei gewöhnlichen Feldern braucht man diesen Verweis durchaus!), kann die Rückgabe einfach ignoriert werden:

```
wdMerge.Fields.Add wdRng, "Anschrift"

wdRng.Fields.Add wdRng, wdFieldMergeField, "Anschrift"

wdDoc.Fields.Add wdRng, wdFieldMergeField, "Anschrift"
```

Welche dieser drei Methoden eingesetzt wird, ist vollkommen gleichgültig. Das spricht für die erste, weil sie am kürzesten ist.

```
Set wdRng = wdDoc.Bookmarks("Betreff").Range
wdRng.Text = "Ihre Kundennummer"
```

Der Betreff ist für alle Adressaten gleich, daher kein SD-Feld.

```
Set wdRng = wdDoc.Bookmarks("Anrede").Range
wdMerge.Fields.Add wdRng, "TextAnrede"
```

Hier wurde angenommen, die Textmarke im Dokument heiße ‚Anrede', die Spalte der *Datenquelle*, die an dieser Stelle als Seriendruckfeld eingefügt werden soll, heiße aber ‚TextAnrede'. Diesen *letzteren* Namen **muß** das Seriendruckfeld bekommen!

```
Set wdRng = wdDoc.Bookmarks("Text").Range

wdRng.Text = "Ihre neue Kundennummer lautet "
```

Bis hierhin ist der Text wieder konstant. Hinter diesen Satz soll ein Feld für die neue Kundennummer eingefügt werden.

```
wdRng.Collapse wdCollapseEnd
```

Mit der Collapse-Methode wird der Bereich auf eine Einfügemarke hinter dem ursprünglichen Ende zusammengezogen. Genau hierhin soll das Feld.

```
wdMerge.Fields.Add wdRng, "idKd"

wdMerge.ViewMailMergeFieldCodes = False
```

Diese Eigenschaft regelt, ob im Seriendruckdokument Feldplatzhalter oder Feldinhalte angezeigt werden.

```
wdMerge.Destination = wdSendToNewDocument
'wdMerge.Destination = wdSendToPrinter
```

Hier wurde entschieden, wie die Seriendruckausgabe erfolgen soll. Wir wählen ein neues Dokument. Für umfangreiche Mailings wäre die direkte Ausgabe auf den Drucker besser, da schneller.

```
wdMerge.Execute
```

Hiermit wird der Seriendruck ausgeführt.

```
Set wdApp = wdDoc.Application
Set wdDocSDAusgabe = wdApp.Documents(1)
```

Das *zuletzt erzeugte* Dokument hat den Index 1. Daher sollte man **sofort** eine Zuweisung an eine Variable vornehmen. Die Execute-Methode hat leider keinen Rückgabewert.

```
Debug.Print wdDocSDAusgabe.Name

wdApp.Visible = True
```

Ein Problem hat man, wenn man in einem Seriendruck 1:n-Daten ausgeben will. Der Seriendruck erzeugt in jedem Fall je Datensatz der Quelle einen neuen Abschnitt. Wenn man also zum Beispiel fünf Datensätze mit Rechnungspositionen hat, die „zufällig" in der Kundennummer übereinstimmen, bekommt der Kunde fünf Rechnungen mit je einer Position.

Falls man nur eine einfache Liste hat, kann man diese bereits in der Datenbank mit Tabulatoren und Zeilenwechseln als Trennzeichen in einem Feld konkatenieren. Das ist aber allenfalls eine Notlösung.

Eine andere Lösung hat einen ellenlangen Bart und wird bereits in der KB zu Word 97 beschrieben. Hier sollen mit einem Serien**katalog** anstelle eines Serien**briefes** in Abhängigkeit von einem Wechsel im Fremdschlüsselfeld durch eine irrwitzige Verschachtelung abstruser Feldfunktionen Seitenwechsel erzeugt werden, wenn sich der Fremdschlüssel ändert. Das Vorgehen ist allerdings freundlich gesagt jenseits von Gut und Böse. Unter der Prämisse, daß eine bereits bestehende Vorlage benutzt werden soll, die obendrein die n-Daten als sauber formatierte Tabelle ausgibt, ist das Verfahren unbrauchbar.

Meine favorisierte Lösung sieht so aus, daß man zunächst nur die Datensätze der 1-Seite – im Rechnungsbeispiel Kundendaten – ausgibt und dabei in der vorgegebenen Leertabelle für die Rechnungspositionen in Zelle 2, 1 ein Feld einfügt, das den Fremdschlüssel, also z. B. die Kundennummer enthält.

Achtung! Zelle 1, 1 kann nicht verwendet werden, weil dort im Zweifel bereits die Überschrift der ersten Spalte steht.

Nach der Ausgabe in ein neues Dokument enthält dieses jetzt pro Kundenrechnung je eine leere Tabelle, die in Zelle 2,1 den Kundenschlüssel enthält. Unter der Annahme, daß jeweils auch eine Blindtabelle zum Briefkopfaufbau vorhanden ist, wäre also in jedem *Abschnitt* des neuen Dokuments die zweite Tabelle die, die mit Rechnungspositionen zu versorgen wäre. Falls die interessierende Tabelle eine andere Nummer in der Vorlage hat, beißt einen das auch nicht. Seine eigene Vorlage kennt man ja, und damit auch die Tabellenposition.

Das mit den Abschnitten hat man wörtlich zu nehmen. Word trennt die sich für aufeinanderfolgende Datensätze ergebenden Textteile durch einen sogenannten *Abschnittswechsel, nächste Seite*.

Im Objektmodell heißen die Abschnitte Sections und man kann sie in einer Schleife durchlaufen. Hier liest man dann einfach den Schlüssel aus Tabelle 2, Zelle 2, 1 und baut ein Recordset mit diesem Kriterium auf die n-Daten (Rechnungspositionen) auf. Damit füllt man dann, wie oben gezeigt, die Tabelle.

```
For i = 1 To wdDocSDAusgabe.Sections.Count - 1

    Set wdTab = wdDocSDAusgabe.Sections(i).Range.Tables(2)
    t = wdTab.Cell(2, 1).Range.Text
    idkd = CLng(Left$(t, Len(t) - 2))
    SQL = "SELECT * FROM qrySD2Pos WHERE fiKd=" & idkd
    Set rcs = CurrentDb.OpenRecordset(SQL)
    n = rcs.RecordCount
    s = 0
    For j = 1 To n
        wdTab.Cell(j + 1, 1).Range.Text = _
                                    "" & rcs.Fields(2).Value
```

```
        wdTab.Cell(j + 1, 2).Range.Text = _
                                    "" & rcs.Fields(3).Value
        wdTab.Cell(j + 1, 3).Range.Text = _
                                    "" & rcs.Fields(4).Value
        s = s + Nz(rcs.Fields(5).Value, 0)
        wdTab.Cell(j + 1, 4).Range.Text = _
                                    "" & rcs.Fields(5).Value
        wdTab.Rows.Add
        rcs.MoveNext
    Next j
    rcs.Close

    wdTab.Cell(j + 1, 1).Range.Text = "Summe"
    wdTab.Cell(j + 1, 4).Range.Text = s
    wdTab.Rows(j + 1).Range.Font.Bold = True

Next i
```

Man muß also mitnichten den Text nach irgendwelchen Vergleichswerten durchwühlen, sondern kann die gewünschten Positionen einfach mit Auflistungen des Objektmodells ansteuern.

Noch zwei Erläuterungen:

Warum Sections.Count – 1? Word trennt je zwei Datensätze durch einen *Abschnittswechsel, nächste Seite*. Der letzte Datensatz bekommt aber einen *Abschnittswechsel, fortlaufend* verpaßt, da sonst eine Leerseite hinten anhinge. Hinter dem Ende des Serienbriefes hängt also noch ein leerer Abschnitt der Ausdehnung null, der zwar gezählt wird, aber nichts enthält.

Warum CLng(Left$(t, Len(t) - 2))? Jede Zelle enthält als Abschluß einen Absatz vbCrLf, der von der Text-Eigenschaft mitgelesen wird. Dieses Doppelzeichen muß von der Kundennummer abgetrennt werden.

Ein paar Grundlagen

Als Programmierer neigt man dazu, Text als trocken Brot anzusehen, das ganz jämmerlich mit Leerzeichen, Tabulatoren und Eingabetasten beschmiert wird. Die zarte Ästhetik blumigen Schriftsatzes mit seinen verästelten Feinheiten ist uns groben Knechten der Logik oft fremd.

Da Word aber keine derbe Bauernmagd ist wie unsere Entwicklungsumgebungen, sondern eine feine Dame vom adeligen Stande derer von Textverarbeitungen, sollte man, wenn man mit ihr flirtet, einige Benimmregeln ins Auge fassen.

Einzüge werden nicht durch Übergabe von Tabulatoren oder – oh Graus! – gar durch Leerzeichenorgien erzeugt, sondern durch … Einzüge.

```
With wdDoc.Paragraphs(i)
    .FirstLineIndent = 20
    .LeftIndent = 10
    .RightIndent = 10
```

Von oben nach unten: Erstzeileneinzug, linker Einzug, rechter Einzug.

Leerzeilen zwischen Absätzen werden nicht durch tumbes ENTER (beziehungsweise vbCrLf & vbCrLf & …) erzeugt, sondern heißen Absatzabstände und werden wie folgt angegeben:

```
    .SpaceAfter = 6
    .SpaceBefore = 12
```

Damit werden Abstände in Punkt, die Word nach respektive vor einem Absatz freihält, festgelegt.

Der Zeilenabstand innerhalb eines Absatzes wird wie folgt festgelegt:

```
    .LineSpacingRule = wdLineSpaceAtLeast
    .LineSpacing = LinesToPoints(1.5)
```

Für LineSpacingRule gibt es verschiedene Konstanten – die im Beispiel verwendete ist in der Regel die sinnvollste – mit denen man das genaue Verhalten festlegt; LineSpacing ist dann der eigentliche Abstand, hier anderthalbzeilig, was durch LinesToPoints in 18 Punkt umgerechnet wird. Man kann diesen Wert, wenn man ihn weiß, auch direkt angeben.

Die vertikale Absatzausrichtung (zentriert, Blocksatz, linksbündig, rechtsbündig) wird durch folgende Eigenschaft angegeben:

```
    .Alignment = wdAlignParagraphCenter
    .Alignment = wdAlignParagraphJustify
    .Alignment = wdAlignParagraphLeft
    .Alignment = wdAlignParagraphRight
```

Das Verhalten von Absätzen beim Seitenwechsel wird mit folgenden Eigenschaften gesteuert:

```
    .PageBreakBefore = True
    .KeepTogether = True
    .KeepWithNext = True
```

Erste Code-Zeile: Seitenwechsel vor diesem Absatz, das heißt, der Absatz beginnt immer auf einer neuen Seite.

Zweite Code-Zeile: Der Absatz darf am Seitenende nicht zerrissen werden. Wenn er nicht ganz auf die aktuelle Seite paßt, wird er komplett auf die nächste übernommen.

Dritte Code-Zeile: Der Absatz und sein Nachfolger müssen auf einer Seite beisammenbleiben. Wenn nicht beide Absätze unten auf eine Seite passen, werden beide auf den Anfang der Folgeseite übernommen.

```
    .Hyphenation = True|False
End With
```

Für diesen Absatz automatische Silbentrennung verwenden: Ja, nein.

Ein geschütztes Leerzeichen (an diesem erfolgt kein Zeilenumbruch; zum Beispiel wünschenswert zwischen Straße und Hausnummer) wird mit Chr$(160) übergeben.

Ein Bindestrich ist kein Trennstrich. Den Trennstrich übergibt man mit Chr$(31) Ein Trennstrich wird nur sichtbar, wenn er durch den Umbruch am Zeilenende in die Silbentrennzone gerät.

Ein Bindestrich ist auch kein Gedankenstrich. Den Gedankenstrich übergibt man mit Chr$(150) Er ist länger als ein Bindestrich.

Einen Seitenwechsel kann man mit Chr$(12) erzwingen.

Chr$(13) oder vbCrLf – Word schluckt beides – ist ein Absatzwechsel.

Ein Zeilenwechsel innerhalb eines Absatzes wird mit Chr$(11) eingegeben. Das wird auch weiche Zeilenschaltung genannt.

Text richtet man nicht mit Tabulatoren aus, sondern mit der Tabellenfunktion von Word. Das ist erheblich gesünder für den Textfluß.

Excel

Arten von Zugriffsmethoden

Es gibt für den Zugriff auf Excel eine Legion von Möglichkeiten.

- Verknüpfte Tabelle
- QueryTable in Excel
- OpenDatabase in Excel
- Abfrage mit Excel-Pfad
- DAO-Recordset mit/ohne SQL
- ADODB-Recordset mit/ohne SQL
- CopyFromRecorset-Methode DAO/ADO
- Automation

Die auf SQL bzw. Recordset basierenden Methoden üben auf den Datenbänker naturgemäß einen eigenen Reiz aus; befindet er sich doch hier auf vertrautem Terrain. Vorgehensweisen, die von vornherein unbrauchbar sind, seien gleich vorneweg genannt:

Eingelinkte Excel-Tabellen sind einer relationalen Datenbank schlicht unwürdig. Da sie mittlerweile schreibgeschützt sind, übt der gemeine User inzwischen oft freiwillig Verzicht. Die Opendatabase-Methode von Excel erstellt grundsätzlich eine neue

Mappe und ist nicht steuerbar, QueryTables sind ebenfalls schlecht anpaßbar. Beide Methoden sind somit letztlich uninteressant.

Recordsets und SQL mit Excel haben in einigen Situationen ihre Berechtigung. Allzu viele Gestaltungsmöglichkeiten hat man aber nicht, da das Format der Daten auf beiden Seiten gleichermaßen relational ist.

Den vollen Umfang an Möglichkeiten, relationale Daten in beliebiger Anordnung auszugeben oder nichtrelationale Daten zum Einlesen aufzubereiten, bevor man sie in eine DB übernimmt, hat man nur mit Automation.

Objektinstanzierung

Auch bei Excel eingangs erst der Code, um eine laufende Excel-Instanz mit Verweis auf eine Arbeitsmappen-Variable zu bekommen.

Methode mit Error oder API

```
Dim xlApp As Excel.Application
Dim xlWbk As Excel.Workbook
Dim tWbk As String

tWbk = "Pfad\und\Name.xls"

'Immer eigene Instanz
Set xlApp = CreateObject("Excel.Application")

'Sonst
'Methode1:
On Error Resume Next
Set xlApp = GetObject(, "Excel.Application")
If Err.Number <> 0 Then _
  Set xlApp = CreateObject("Excel.Application")
On Error GoTo 0

'Methode 1 besser
On Error Resume Next
Set xlApp = GetObject(, "Excel.Application")
If Err.Number = 429 _
  Then
    Err.Clear
    Set xlApp = CreateObject("Excel.Application")
  Else
    If Err.Number > 0 _
      Then
        MsgBox "Fataler Fehler"
       'Weitere Fehlerbehandlung
    End If
End If
On Error GoTo 0

'Methode2:
If FindWindow("XLMain", vbNullString) = 0 _
  Then
```

```
    Set xlApp = CreateObject("Excel.Application")
  Else
    Set xlApp = GetObject(, "Excel.Application")
End If
```

Man muß sich natürlich für eine der Methoden entscheiden, also nicht alle in eine Prozedur packen. Es besteht jetzt in jedem Fall ein gültiges xlApp-Objekt. Damit kann man eine Dokumentvariable erzeugen, auf die es eigentlich ankommt:

```
'Neue, leere Mappe mit (Standard) drei Tabellen
Set xlWbk = xlApp.Workbooks.Add
'Neues Dokument von Vorlage
Set xlWbk = xlApp.Workbooks.Add(tWbk)
'Vorhandenes Dokument öffnen
For Each xlwbk In xlApp.Workbooks
    If xlwbk.FullName = tWbk Then Exit For
Next xlwbk
If xlwbk Is Nothing Then Set xlwbk = xlApp.Workbooks.Open(tWbk)
```

Im letzten Fall muß im Gegensatz zu Word tatsächlich geprüft werden, ob die angeforderte Datei nicht bereits geöffnet ist.

Damit ist das Ziel erreicht, die gewünschte Datei geöffnet verfügbar zu haben und in xlWbk einen Objektverweis darauf zu halten.

Methode mit Dokumentinstanzierung

Leere Mappe, unsichtbar

Das Vorgehen ist ganz analog zu Word.

```
Set xlWbk = CreateObject("Excel.Sheet")
Set xlSht = xlWbk.Worksheets(1)
xlSht.Range("A1").Value = "Hallo, Welt"
xlWbk.SaveAs "Pfad\und\Name.xls"
```

xlWbk.Close ist hier allerdings weder nötig noch möglich. Excel hat die Macke, die Mappe in jedem Fall von alleine sofort zu schließen.

Eine bereits laufende Excel-Instanz bleibt stehen, eine von der Methode erzeugte Instanz wird automatisch terminiert.

Leere Mappe, sichtbar

Der Code, der an dieser Stelle bei Word vorgestellt wurde, sollte theoretisch auch mit Excel funktionieren. Die erzeugte Mappe wird aber von Excel sofort wieder sang- und klanglos geschlossen. Es handelt sich um einen Bug, der vermutlich darauf zurückzuführen ist, daß in alten Excel-Versionen eine Mappe nur ein Blatt haben konnte und daher ein Sheet-Objekt nicht mit einem Workbook-Objekt deckungsgleich ist.

```
Set xlWbk = CreateObject("Excel.Sheet")
Set xlApp = xlWbk.Application
xlWbk.Windows(1).Visible = True
xlApp.Visible = True
AppActivate xlApp.Caption
xlWbk.Activate
```

```
'klappt nicht
```

Es ist aber kein Problem, diesen kleinen Fehler zu flicken.

```
Set xlWbk = CreateObject("Excel.Sheet")
Set xlApp = xlWbk.Application
Set xlWbk = xlApp.Workbooks.Add
xlApp.Visible = True
AppActivate xlApp.Caption
xlWbk.Activate
```

Dieser Code macht genau das Gewünschte.

Vorhandene Mappe bearbeiten, unsichtbar

Das Öffnen vorhandener Mappen geht ganz analog. Der einzige Unterschied ist die erste Zeile, die anstelle des Klassennamens den Dateipfad enthält.

```
Set xlWbk = CreateObject("Pfad\Name.xls")
Set xlSht = xlWbk.Worksheets(1)
xlSht.Range("B5").Value = "Nochmal hallo"
xlWbk.Windows(1).Visible = True '!
xlWbk.Save
xlWbk.Close
```

Anders als bei Word muß vor dem Speichern die Sichtbarkeit der Mappe auf wahr gesetzt werden, da Excel im Gegensatz zu Word diese Sichtbarkeit als Eigenschaft der Datei abspeichert. Eine so erzeugte Mappe, die ohne Visible = True gespeichert wird, wäre beim nächsten Öffnen unsichtbar, was den Benutzer wahrscheinlich etwas verblüffen würde.

Vorhandene Mappe bearbeiten, sichtbar

Hier geht's wieder fast analog zu Word

```
Set xlWbk = CreateObject("Pfad\Name.xls")
Set xlApp = xlWbk.Application
xlWbk.Saved = True '!
'Hier ggf. Manipulation von xlWbk
xlWbk.Windows(1).Visible = True
xlApp.Visible = True
AppActivate xlApp.Caption
xlWbk.Activate
```

Direkt nach dem Öffnen ist die Mappe auf Saved = True zu setzen. Wenn das unterbleibt, kann es passieren, daß eine Speicher-rückfrage kommt, obwohl am Dokument nichts geändert wurde. Durch Saved = True wird die Mappe nicht wirklich gespeichert, sie bekommt nur vorgegaukelt, daß seit dem letzten Speichern nichts geändert worden wäre. Wenn nach dieser Zeile weitere Änderungen stattfinden, wird die Mappe wieder Saved = False, analog zu einem Datensatz der schmutzig wird.

Vorhandene Mappe als Vorlage, sichtbar

Auch hier kann die bei Word vorgestellte Methode angepaßt werden.

```
FileCopy "Pfad\Vorlage.xls", "Pfad\NeuDokument.xls",
Set xlWbk = CreateObject("Pfad\NeuDokument.xls")
Set xlApp = xlWbk.Application
xlWbk.Saved = True
'Hier ggf. Manipulation von xlWbk
xlWbk.Windows(1).Visible = True
xlApp.Visible = True
AppActivate xlApp.Caption
xlWbk.Activate
```

Falls jemand Vorbehalte gegen die Create-Methoden mit Dokumenten haben sollte und lieber von einer Application-Instanz ausgehen möchte, empfiehlt sich für Excel dennoch das folgende Vorgehen, das das oben erwähnte seltsame Verhalten bei CreateObject bewußt ausnutzt.

```
Set xlWbk = CreateObject("Excel.Sheet")
Set xlApp = xlWbk.Application

'Neue, leere Mappe mit (Standard) drei Tabellen
Set xlWbk = xlApp.Workbooks.Add
'Neues Dokument von Vorlage
Set xlWbk = xlApp.Workbooks.Add(tWbk)
'Vorhandenes Dokument öffnen
For Each xlwbk In xlApp.Workbooks
    If xlwbk.FullName = tWbk Then Exit For
Next xlwbk
If xlwbk Is Nothing Then Set xlwbk = xlApp.Workbooks.Open(tWbk)
```

Dieses Vorgehen leistet ebenfalls alles, was bei der Anwendungsinstanzierung viel Aufwand erfordert mit nur zwei Zeilen.

Echte Fans von kurzen Codes können das sogar in einen Einzeiler umformen:

```
Set xlWbk = _
        CreateObject("Excel.Sheet").Application.Workbooks.Add
```

Obendrein leistet diese Methode noch etwas, das CreateObject("Excel.Application") *nicht* macht: Es werden *sämtliche aktivierten Add-Ins* sowie die Datei *Personl.xls* geladen, so daß der Benutzer sein Excel genau so vorfindet, wie er es sich eingerichtet hat. Bei der Instanzierung über CreateObject("Excel.Application") bekommt man nämlich ein sehr karges Excel, das all diese Add-Ins nicht mitgeladen hat. Man müßte diese ggf. alle per Code nachladen, was nicht ganz ohne ist und in verschiedenen Excel-Versionen auch aus verschiedenen Registry-Zweigen zusammengeklaubt werden muß.

Für welche Methode man sich auch immer entscheidet, hat man an dieser Stelle nun einen gültigen Verweis auf die gewünschte Mappe.

Die wichtigsten Excel-Objekte

- Application
    - Names(i)

- Workbooks(i)
  - Names(i)
  - WorkSheets(i)
    - Names(i)
      - *RefersTo*
    - **Range**
      - *Range (!)*
      - *Resize*
      - *Offset*
      - *EntireRow*
      - *EntireColumn*
      - Interior
      - Font
      - Borders
    - *Rows*
    - *Columns*

Kursive Elemente sind alle vom Typ Range

Application bezeichnet wie immer die Anwendungsinstanz, hier das laufende Excel.

Workbooks sind alle geöffneten Arbeitsmappen, das Pendant zu den Documents bei Word. Auch hier gilt, daß man einen Objektbezug auf eine bestimmte Arbeitsmappe in der Regel bereits beim Öffnen erzeugt.

Worksheets sind die Tabellenblätter einer Arbeitsmappe. Bezogen auf die Objekthierarchie liegen diese wie eine weitere Schicht zwischen dem Dateiobjekt und den Daten.

- Word: Document.Range.Text
- Ecxel: Workbook.Worksheet.Range.Value

Die **Names**-Auflistung repräsentiert benannte Bereiche. Gleichnamige Auflistungen stehen auf drei Ebenen zur Verfügung:

- Application.Names: Alle Namen aller Tabellen in allen Mappen
- Workbook.Names: Alle Namen aller Tabellen der angegeben Mappe
- Worksheet.Names: Alle Namen nur in der angegebenen Tabelle

Da Excel im Gegensatz zu Word eine einfache Adressierungsmöglichkeit kennt, kann man einen Range direkt aus dem

Worksheet durch Bereichsangabe gewinnen. Für ein Analogon zu Bookmarks besteht kein Bedarf. Trotzdem gibt es das natürlich. Es sind eben die benannten Bereiche (englisch: Names).

Range steht für einen Zellbereich. Das kann eine einzelne Zelle eine Gruppe von Zellen oder eine ganze Tabelle sein. Wie schon bei Word ist auch in Excel das Range-Objekt der Schlüssel zum Verständnis effizienter Programmierung.

Probleme des Makro-Rekorders

Der Makro-Recorder arbeitet mit den Objekten ActiveWorkbook, ActiveSheet, ActiveCell und Selection.

Typischer Rekorder-Code sieht z. B. so aus:

```
    Sheets("Tabelle3").Select
    Range("C6").Select
    ActiveCell.FormulaR1C1 = "1"
    Range("C7").Select
    ActiveCell.FormulaR1C1 = "2"
    Range("C8").Select
    ActiveCell.FormulaR1C1 = "3"
    Range("C9").Select
    ActiveCell.FormulaR1C1 = "4"
    Range("C10").Select
    ActiveCell.FormulaR1C1 = "5"
    Range("C11").Select
```

Dieser Code ist völlig ungeeignet für Automation. Er ist extrem langsam und – viel schlimmer – fehleranfällig. Als Default-Mutterobjekt für Sheets gilt ActiveWorkbook. Wenn während des Programmablaufs eine andere Mappe aktiviert wird, landen die Daten ab diesem Zeitpunkt in ebendieser. Ähnlich ist Range hier vom Default ActiveSheet abgeleitet und ActiveCell bezieht sich auf die Selection. Ein einziger Mausklick kann genügen, um völlig absurde Ausgaben zu erzeugen. Und ja, der Benutzer oder anderer ähnlich genialer Code hat genug Zeit zu interferieren. Große Datenmengen würden mit solchem Vorgehen in vielen Minuten statt in Millisekunden übertragen.

Über die Zuweisung vom gewünschten Workbook und dem auszuwählenden WorkSheet an Objekt-Variablen und Arbeit mit dem Range-Variablen wird Zugriffssicherheit erzielt, die unabhängig von Selektionen und obendrein um Größenordnungen schneller ist.

Worksheets handhaben

Die benötigten Operationen mit Worksheet-Objekten sind sehr überschaubar. Den umfangreichsten Anteil an der Daten-manipulation hat das im Anschluß besprochene Range-Objekt.

Anzahl Tabellenblätter

```
Debug.Print xlWbk.Worksheets.Count
```

Neues Tabellenblatt hinten anfügen

```
Set xlSht = xlWbk.Worksheets.Add(, _
                    xlWbk.Worksheets(xlWbk.Worksheets.Count), 1)
xlSht.Name = "NeuesBlatt"
```

Neues Tabellenblatt vorne einfügen

```
Set xlSht = xlWbk.Worksheets.Add(xlWbk.Worksheets(1), , 1)
xlSht.Name = "NeuesBlatt"
```

Alternative Syntax

```
xlWbk.Worksheets.Add.Name = "NeuesBlatt"
Set xlSht = xlWbk.Worksheets("NeuesBlatt")
```

Blatt löschen

```
'Drittes Blatt
xlWbk.Worksheets(3).Delete

'Blatt "Blattname"
xlWbk.Worksheets("Blattname").Delete

'Blatt, das von Objektvariable xlSht referenziert wird
xlSht.Delete

'Alle Blätter bis auf letztes löschen
For i = 1 To xlWbk.Worksheets.Count - 1
    xlWbk.Worksheets(1).Delete
Next i

'Alle Blätter bis auf erstes löschen
For i = 1 To xlWbk.Worksheets.Count - 1
    xlWbk.Worksheets(2).Delete
Next i
```

Das Range-Objekt von Excel

Trotz der Namensgleichheit und gewisser Ähnlichkeiten bestehen zwischen Word-Ranges und Excel-Ranges erhebliche Unterschiede.

Bei Word haben viele anschauliche Objekte wie Absatz, Tabelle, Textmarke und weitere eine Range-Eigenschaft, die einen Bezug auf ein Range-Objekt, das die Ausdehnung des Mutterobjekts repräsentiert, herstellen.

Bei Excel ist das eher die Ausnahme. Es gibt Dutzende von Eigenschaften, die nicht Range heißen, aber eine Referenz auf ein Range-Objekt ergeben.

Erzeugen von Ranges

Im folgenden ist vorausgesetzt, daß xlSht einen Objektverweis auf eine interessierende Tabelle in einem bestimmten Workbook enthält.

```
Set xlSht = xlWbk.Worksheets("Tabelle1")
```

Dann geben folgende Methoden jeweils ein Range-Objekt der angegebenen Ausdehnung zurück. Bei Syntaxvarianten ist die vorzuziehende fett markiert.

Ganzes Tabellenblatt

```
Set xlRng = xlSht.Cells
```

Eine bestimmte Zelle, im Beispiel C5

```
Set xlRng = xlSht.Range("C5")
Set xlRng = xlSht.Cells(5, 3)
```

Man beachte, daß bei Cells die Reihenfolge der Zeilen- und Spaltenbezüge vertauscht ist: C: Spalte, 5: Zeile, aber 5: Zeile, 3: Spalte

Rechteck mit oberer linker und unterer rechter Ecke, A1 bis D12

```
Set xlRng = xlSht.Range("A1:D12")
Set xlRng = xlSht.Range(xlSht.Cells(1, 1), xlSht.Cells(12, 4))
```

Dieser Fall ist der in der Praxis am häufigsten auftretende. Die Syntax mit Cells mag hier etwas schwerfällig erscheinen, zumal A1:D12 erheblich anschaulicher ist. Man sollte sich aber von vornherein, auch wenn die Versuchung von A1 groß ist, angewöhnen, nur mit der Cells-Syntax zu arbeiten. Wenn man generische Funktionen schreibt, wird man haufenweise mit Zeilen- und Spaltennummern konfrontiert, die sich aus der Anwendungslogik ergeben und dann direkt eingesetzt werden können.

3 Spalten weiter rechts als sonst ist schlicht Spaltennummer plus drei. Bei Spalte BY (ja, nach Z kommen Doppelbuchstaben) viel Spaß beim Ermitteln von CB. Merke: Zahlen sind Freunde.

Eine ganze Spalte, hier C

```
Set xlRng = xlSht.Range("C:C")
Set xlRng = xlSht.Columns(3)
```

Eine ganze Zeile, hier die vierte

```
Set xlRng = xlSht.Range("4:4")
Set xlRng = xlSht.Rows(4)
```

Aus Gründen der Anschaulichkeit wird nachfolgend die böse Syntax Range("A1") oder Range("A1:C3") statt Cells(1,1) oder Range(.Cells(1,1), .Cells(3,3)) verwendet. Im wahren Prozedurleben bitte stattdessen Cells verwenden!

Gesamte Spalte oder Zeile, die eine Zelle enthält, zurückgeben:

```
Set xlRng = xlSht.Range("C5").EntireColumn
'Spalte C
Set xlRng = xlSht.Range("C5").EntireRow
'Zeile 5
```

Das geht auch mit Zellblöcken:

```
Set xlRng = xlSht.Range("C5:D7").EntireColumn
'Spalten C - D
Set xlRng = xlSht.Range("C5:D7").EntireRow
```

```
'Zeilen 5 - 7
```

Ganz wichtig ist die folgende Resize-Methode. Damit kann man einen Range einer angegeben Höhe und Breite erzeugen. Wenn man zum Beispiel beginnend in Zelle C5 eine Breite entsprechend der Feldanzahl eines Recordsets und eine Höhe entsprechend der Datensatzanzahl braucht, ist es erheblich bequemer, ausgehend von C5 mit diesen beiden Zahlen den Bereich aufzubauen als die letzte Zelle zu berechnen.

```
Set xlRng = xlSht.Range("C5").Resize(8, 4)
Set xlRng = xlSht.Cells(5, 3).Resize(8, 4)
'Vergrößern/verkleinern auf 8 Zeilen, 4 Spalten, ausgehend von
'oberer linker Ecke
```

Die Syntax ist obendrein wesentlich handlicher, als Range mit oberer linker und unterer rechter Ecke. *Diese Methode ist daher der Königsweg, um einen Range zu referenzieren.*

Typisches Recordset-Beispiel:

```
Set xlRng = xlSht.Cells(5, 3)
r = rcs.RecordCount
c = rcs.Fields.Count
Set xlRng = xlRng.Resize(r, c)
```

Auch Blöcke können vergrößert und verkleinert werden. Hier ein Beispiel, wie das relativ statt wie eben absolut geschieht:

```
Set xlRng = xlSht.Range("C5:E12").Resize(xlRng.Rows.Count + 8,_
                                         xlRng.Columns.Count + 4)
'Vergrößern um 8 Zeilen, 4 Spalten
```

Mit der Offset-Methode wird ein Range unter Beibehaltung seiner Ausdehnung verschoben:

```
Set xlRng = xlSht.Range("C5:E12")
Set xlRng = xlRng.Offset(3, 5)
```

Damit wurde der Range um 5 nach rechts und 3 nach unten verschoben. xlRng bezieht sich jetzt auf den Bereich H10:J17

```
Set xlRng = xlSht.Range("A1")
Set xlRng = xlRng.Offset(1, 0)
```

Dadurch wird ein Range um 1 nach unten verschoben. Wenn der Ausgangs-Range wie im Beispiel sich auf nur eine Zelle bezieht, wandert dadurch die bezogene Zelle sukzessive nach unten, was ein bißchen wie MoveNext bei Datensätzen ist. Das könnte bei Datenbänkern falsche Sympathien erwecken. Die sich in den Vordergrund drängende Idee, in einer Recordset-Schleife mit dem MoveNext parallel einen Range mittels Offset mitwandern zu lassen, um dort jeweils Daten einzuschreiben, ist eine ganz und gar schlechte. Offset ist für große Datenmengen aus verschiedenen Gründen fürchterlich langsam. Die sinnvollste Methode, das Befüllen eines Range mittels Arrays, wird weiter unten beschrieben.

```
Set xlRng = xlSht.Names("Test").RefersTo()
```

Die RefersTo-Methode ergibt einen Objektverweis auf den Range, der von einem definierten Namen belegt ist. Die obige Anweisung hätte in Word ihr Pendant in

```
Set wdRng = wdDoc.Bookmarks("Test").Range
```

Da man in einer Arbeitsmappe aber über Tabellenblattname und Zelladressierung bereits über eine wohldefinierte numerische Zugriffsmöglichkeit auf bestimmte Bereiche verfügt, ist die Verwendung von Bereichsnamen von erheblich geringerer Bedeutung als in Word.

Range aus Range

Das Range-Objekt verfügt selbst wieder über eine Range-Methode. Damit kann man relative Unterbereiche ableiten.

```
Set xlRng = xlSht.Range("C5:G12")
Debug.Print xlRng.Address
'> $C$5:$G$12
xlRng.BorderAround , xlMedium

Set xlRngChild = xlRng.Cells(1, 1)
Debug.Print xlRngChild.Address
'> $C$5
xlRngChild.Interior.Color = vbRed

Set xlRngChild = xlRng.Range("B1:C2")
Debug.Print xlRngChild.Address
'> $D$5:$E$6
'Relativ von C5 aus gesehen ist B1 im Blatt D5 und C2 E6
xlRngChild.Interior.Color = vbYellow

Set xlRngChild = xlRng.Cells(2, 3).Resize(3, 4)
Debug.Print xlRngChild.Address
'> $E$6:$H$8
xlRngChild.Interior.Color = vbBlue
```

Arbeiten mit dem Range

Es ist schön, wenn man Ranges festlegen kann und es wurden etliche Methoden gezeigt, die einen Objektverweis auf einen Range erzeugen, aber dann möchte man natürlich auch etwas Sinnvolles damit machen.

Die wichtigste Range-Eigenschaft ist Value. Damit kann ein Range mit Daten befüllt werden.

```
Set xlRng = xlSht.Range("A1")
xlRng.Value = "Hallo"
'Zelle A1 enthält jetzt den Text "Hallo" ohne Anführungszeichen
xlRng.Value = 5
'Zelle A1 enthält jetzt die Zahl 5
```

Das Auslesen geht ganz analog.

```
Dim v As Variant
v = xlRng.Value
```

Hier stellt sich die Frage, was passiert, wenn der Range über mehrere Zellen geht.

```
Set xlRng = xlSht.Range("A1:C7")
xlRng.Value = 5
```

Das Ergebnis ist, daß alle Zellen von A1 bis C7 jeweils die Zahl 5 enthalten. Viel interessanter wäre natürlich, auf diese Art mehrere verschiedene Werte mit einem Schritt zuweisen zu können.

Nun – man kann. Man muß einfach nur ein Array verwenden:

```
Dim v(1 To 3, 1 To 7) As Variant 'auch String, Long etc.
For i = 1 To 3
    For j = 1 To 7
        v(i, j) = 10 * i + j
    Next j
Next i
xlRng.Value = v()
```

Es dürfte klar sein, daß es von entscheidender Bedeutung ist, daß der Range und das Array dieselben Dimensionen haben.

```
Set xlRng = xlSht.Range("B3")
'gewünschte Startzelle
Set xlRng = xlRng.Resize(UBound(v, 1), Ubound(v, 2))
'Bereich auf gleiche Größe wie Array bringen
xlRng.Value = v()
```

Die Array-Verwendung funktioniert auch beim Auslesen:

```
Dim v As Variant 'Normaler Variant, kein Array
v = xlRng.Value
If IsArray(v) _
    Then
        For i = 1 To Ubound(v, 1)
            For j = 1 To UBound(v, 2)
                Debug.Print v(i, j)
            Next j
        Next i
    Else
        Debug.Print v
End If
```

Bei diesem Vorgehen liefert Excel bei einem Range, der mehr als eine Zelle umfaßt, ein zweidimensionales Variant-Array mit eins als kleinstem Index zurück. Das ist auch der Fall, wenn der Range nur eine Spalte oder nur eine Zeile hat:

```
Dim v As Variant

Set xlRng = xlSht.Range("A1:A12")
v = xlRng.Value
```

Man könnte jetzt erwarten, daß v ein Array (1 To 12) ist. Dem ist aber nicht so. Vielmehr ist v (1 To 12, 1 To 1) und muß mit

```
v(i, 1)
```

statt mit

```
v(i)
```

durchlaufen werden. Excel ist zwar in der Lage, ein eindimensionales Array vom Typ (1 To 12) zu schreiben, aber nur zeilenweise. Bei Spalten wiederholt sich der erste Array-Wert.

Man sollte also immer einsbasierte zweidimensionale Arrays verwenden, selbst wenn 1 To 1 etwas seltsam anmutet.

Falls der Range sich auf nur eine Zelle bezieht, liefert der Code

```
v = xlRng.Value
```

allerdings kein Array (1 To 1, 1 To 1), sondern einen einfachen Wert, daher auch oben die Prüfung auf IsArray.

Es gibt auch einen Fall, in dem ein Range mehrere Zellen hat, aber xlRng.Value dennoch ein Wert und kein Array ist. Das kann bei einem Union-Range passieren, dessen erster Bereich nur eine Zelle enthält.

Ein Union-Range ist ein nichtrechteckiger Zellbereich, den man aus mehreren Rechteckbereichen zusammenstellen kann. Ein Beispiel dazu später. Um abzusichern, daß ein Range ein einfacher Rechteckbereich ist, kann folgende Prüfung verwendet werden:

```
xlRng.Areas.Count = 1
```

Da sowohl das Lesen oder Schreiben aller Daten eines Bereichs mit einem einzigen Schritt als auch das Aufbauen eines Arrays im RAM außerordentlich schnelle Operationen sind, ist diese Vorgehensweise unschlagbar performant. Der Erzeugung nichtrelationaler, d. h. beliebig angeordneter Array-Strukturen aus der relationalen Tabellenstruktur eines Recordsets ist das nächste Kapitel vollständig gewidmet.

Da man einen Bericht in Form eines Tabellenblattes oft auch mit Formeln wird ergänzen wollen, ist es an der Zeit, die Formula-Eigenschaft vorzustellen.

```
xlRng.Formula = "=2+3"
```

Damit wird die entsprechende Formel in die Zellen des Ranges eingetragen.

Genau denselben Effekt könnte man mit

```
xlRng.Value = "=2+3"
xlRng.FormulaLocal = "=2+3"
xlRng.FormulaR1C1 = "=2+3"
xlRng.FormulaR1C1Local = "=2+3"
```

erzielen.

Bei einer Formel, die an der Exceloberfläche wie SUMME(A:A) aussähe – das ist die Summe über die gesamte Spalte A – scheiden sich aber die Geister. Zu allem Überfluß unterscheiden sich die fünf Varianten auch noch beim Lesen und Schreiben im Verhalten.

Schreiben:

Value: Damit wird der Wert einer Zelle gesetzt oder gelesen. Beim Schreiben könnte der Wert ein Text sein, der eben zufällig von Excel als Formel interpretierbar ist.

```
Set xlRng = xlSht.Range("B1")
xlRng.Value = "=Sum(A:A)"
xlRng.Value = "=Sum(C[-1])"
```

In einem deutschen Excel erscheint in beiden Fällen in der Zelle der Eintrag =SUMME(A:A), der auch entsprechend funktioniert.

```
xlRng.Value = "=SUMME(A:A)"
```

führt aber zu einem Fehler! Es wird zwar in der Zelle korrekt =SUMME(A:A) eingetragen, aber das wird von Excel nicht als Formel erkannt und statt des Ergebnisses wird #NAME? angezeigt.

Formula schreibt eine Formel in englischer Sprache. Dabei kann die A1-Bezugsart oder die weniger anschauliche, aber leistungsfähigere Z1S1-Bezugsart verwendet werden.

Beim Schreiben verhält sich Formula also genau wie Value, d. h.

```
Set xlRng = xlSht.Range("B1")
xlRng.Formula = "=Sum(A:A)"
xlRng.Formula = "=Sum(C[-1])"
```

sind erlaubte Varianten, deutsche Funktionsnamen ergeben Fehler.

FormulaR1C1 läßt nur die Syntax

```
xlRng.FormulaR1C1 = "=Sum(C[-1])"
```

zu.

Analog verhält sich FormulaLocal und FormulaR1C1Local. Bei beiden müssen allerdings die deutschen (oder holländischen, tschechischen, kongolesischen oder was auch immer für ein Windows man installiert hat) Funktionsnamen verwendet werden.

Erlaubt ist also:

```
Set xlRng = xlSht.Range("B1")
xlRng.FormulaLocal = "=Summe(A:A)"
xlRng.FormulaLocal = "=Summe(S(-1))"
xlRng.FormulaR1C1Local= "=Summe(S(-1))"
```

Lesen:

Hier werden die vielen verschiedenen Varianten erst sinnvoll:

Angenommen in Zelle B1 steht bereits die Formel =SUMME(A:A) in einem deutschen Excel. Dann liefern die fünf genannten Eigenschaften folgende Ergebnisse:

Value: 68735154 (falls das die Summe aller Zahlen in Spalte A ist)

Formula: =SUM(A:A)

FormulaLocal: =SUMME(A:A)

FormulaR1C1: =SUM(C[-1])

FormulaR1C1Local: =SUMME(S(-1))

Value liefert also das Ergebnis der Formel, die vier Formula-Eigenschaften den Formelausdruck in der jeweils angeforderten Syntax.

Welche Variante sollte man nun nehmen, um Formeln in ein Blatt einzutragen?

Am anschaulichsten ist sicher FormulaLocal, da man hier die vertrauten deutschen Funktionsnamen und die gewohnte A1-Schreibweise verwenden kann. Die schlechte Nachricht: Am besten ist die Verwendung von FormulaR1C1, also englische Funktionsnamen in Z1S1-Schreibweise.

FormulaLocal= "=SUMME(A:A)" würde in einem französischen Windows Fehler verursachen, da es dort FormulaLocal= "=S**O**MME(A:A)" heißen müßte. Formula = "=SUM(A:A)" funktioniert hingegen immer.

Die unanschauliche Z1S1-Bezugsart hat wiederum den entscheidenden Vorteil, wesentlich flexibler zu sein. Da hier relative Abstände zwischen Zellen rein numerisch angegeben werden, gilt das gleiche, was schon eingangs bei der Festlegung von Ranges gesagt wurde. *Zahlen* sind Freunde, nicht Buchstaben.

Man sollte also anstreben, mittels FormulaR1C1 zu programmieren.

Achtung: Sum(C[-1]) versus Summe(S(-1)) ist kein Tippfehler; im Englischen werden eckige Klammern benutzt, im Deutschen runde, um eine relative Adressierung anzuzeigen.

Wenn man als in Deutschland tätiger Programmierer verständlicherweise mit den englischen Formelbezeichnungen Probleme hat, gibt es zwei Hilfsmittel. Zum einen gibt es eine Datei namens VBAListe.xls, die in der Regel auf Excel- oder Office-DVDs herumgeistert oder sich zur Not im Internet findet. Bei einer Standardinstallation von Office 2003 sollte sie sich im Ordner C:\Programme\Microsoft Office\OFFICE11\1031 finden. In dieser Datei enthält das zweite Blatt „Tabellenfunktionen“ die deutschen und englischen Funktionsnamenentsprechungen aufgelistet. Die zweite Möglichkeit, wenn man nur für eine einzelne deutsche Funktion den englischen Namen benötigt, wäre auf einem deutschen System folgender Code:

```
Set xlRng = xlSht.Cells(1, 1)
xlRng.FormulaLocal = "=Summe(B1:B5)"
Debug.Print xlRng.Formula
Debug.Print xlRng.FormulaR1C1
```

Anstelle der Summe kann man jede andere Formel in deutscher Schreibweise eingeben und sich den gewünschten englischen Code in A1- oder R1C1-Schreibweise ausgeben lassen.

Ein häufiges Problem, weniger wenn man Daten ausgibt, oft aber beim Einlesen von Excel-Dateien, ist, daß man die letzte Spalte oder

Zeile bestimmen muß, die noch Daten enthält. Hier wird häufig mit Find und FindNext gearbeitet, was aber langsam ist und – viel schlimmer – durch Leerzellen im zu durchsuchenden Bereich aus dem Tritt gebracht wird. Mit folgendem Code findet man schnell und sicher die letzten beschriebenen Zellen.

Unterste beschriebene Zelle einer Spalte:

```
Public Property Get LastRow( _
                    xlSht As Excel.Worksheet, _
                    Optional Col As Long = &H1&) _
                As Long
If Col < 1 Then Col = 1
If Col > &H100& Then Col = &H100&
LastRow = xlSht.Cells(&H10000, Col).End(xlUp).Row
End Property
```

Rechtsäußerste beschriebene Zelle einer Zeile

```
Public Property Get LastCol( _
                    xlSht As Excel.Worksheet, _
                    Optional Row As Long = &H1&) _
                As Long

If Row < 1 Then Row = 1
If Row > &H10000 Then Row = &H10000
LastCol = xlSht.Cells(Row, &H100&).End(xlToLeft).Column
End Property
```

Letzte beschriebene Zelle eines Rechteckbereichs

```
Public Property Get LastCorner( _
                    xlSht As Excel.Worksheet) _
                As Range

Dim i As Long, r As Long, c As Long, t As Long

For i = 1 To 256    '&H100&
    t = xlSht.Cells(&H10000, i).End(xlUp).Row
    If t > r Then r = t
    If t > 1 Then c = i
Next i

Set LastCorner = xlSht.Cells(r, c)

End Property
```

Das ganze ist das Ergebnis einer so abstrusen Kundenanforderung, daß sie fast schon wieder alltäglich ist: In einer in eine Datenbank einzulesenden Excel-Datei können beliebig viele Spalten befüllt sein, wobei jederzeit ohne feste Regel Leerspalten zwischen Datenspalten auftauchen können. Es muß sichergestellt sein, daß die letzte beschriebene Spalte sicher gefunden wird. Da die Datei umfangreich ist, soll der Einlesevorgang möglichst schnell sein.

Ausgabe mit Variant-Arrays

Das grundsätzliche Vorgehen ist stets dasselbe: Man fragt die Datensätze in der gewünschten Sortierung ab. Das resultierende

Recordset wird in ein zweidimensionales Variant-Array umgeschrieben.

Dieses Delinearisieren von Vektoren (Felder des Recordset) in eine Matrix (zweidimensionales Array) erfolgt, falls keine besonderen inhaltlichen Gründe für eine abweichende Anordnung bestehen, nach dem Grundschema, daß die Datensatznummer (oder ein Index eines eindimensionalen Arrays) – es genügt auch eine mitlaufende Zählvariable – die Indizes des zweidimensionalen Arrays mittels Mod und \ ergibt.

Dabei ist für Excel der erste Index die Zeilennummer, der zweite Index die Spaltennummer.

Um zum Beispiel 12 Werte geordnet in ein 3-auf-4-Schema zu verteilen, gibt es vier Möglichkeiten:

3 Zeilen, 4 Spalten, N-Form (wenn auch spiegelverkehrt)

Eins	Vier	Sieben	Zehn
Zwei	Fünf	Acht	Elf
Drei	Sechs	Neun	Zwölf

3 Zeilen, 4 Spalten, Z-Form

Eins	Zwei	Drei	Vier
Fünf	Sechs	Sieben	Acht
Neun	Zehn	Elf	Zwölf

4 Zeilen, 3 Spalten, N-Form

Eins	Fünf	Neun
Zwei	Sechs	Zehn
Drei	Sieben	Elf
Vier	Acht	Zwölf

4 Zeilen, 3 Spalten, Z-Form

Eins	Zwei	Drei
Vier	Fünf	Sechs
Sieben	Acht	Neun
Zehn	Elf	Zwölf

Wenn wir annehmen, daß die Werte Eins bis Zwölf aufsteigend mit den Indizes oder Datensatznummern 1 bis 12 daherkommen, müssen also aus dieser Zahlenreihe folgende Indexpaare für die entsprechenden Arrays erzeugt werden:

	3x4, N		4x3, N		3x4, Z		4x3, Z	
i	**z**	**s**	**z**	**s**	**z**	**s**	**z**	**s**
1	1	1	1	1	1	1	1	1
2	2	1	2	1	1	2	1	2
3	3	1	3	1	1	3	1	3
4	1	2	4	1	1	4	2	1
5	2	2	1	2	2	1	2	2
6	3	2	2	2	2	2	2	3
7	1	3	3	2	2	3	3	1
8	2	3	4	2	2	4	3	2
9	3	3	1	3	3	1	3	3
10	1	4	2	3	3	2	4	1
11	2	4	3	3	3	3	4	2
12	3	4	4	3	3	4	4	3

Die Zahlenmuster dürften auf Anhieb einleuchten (gut, vielleicht nicht auf Anhieb, aber wenigstens irgendwann); erzeugen kann man sie im Code mit den Operatoren Mod und \

Dabei erzeugt der Mod-Operator immer wieder dieselbe aufsteigende Zahlenfolge, der \-Operator hingegen jeweils einen Stapel derselben Zahl in immer derselben Größe, wobei die Stapel als ganzes hochzählen.

```
For i = 0 To 11
    Debug.Print i Mod 3; i \ 3
Next i
```

```
Mod \
 0  0
 1  0
 2  0
 0  1
 1  1
 2  1
 0  2
 1  2
 2  2
 0  3
 1  3
 2  3
```

Für die 1-basierte Excel-Ausgabe brauchen wir die Zahlen jeweils mit 1 beginnend, was sich problemlos erreichen läßt mit

```
For i = 1 To 12
    Debug.Print (i - 1) Mod 3 + 1, (i - 1) \ 3 + 1
Next i
```

Mit einem Recordset als Quelle könnte das ganze so aussehen:

```
i = 0
With rs
    Do Until .EOF
        i = i + 1
        'oder i = .AbsolutePosition
        Ziel((i - 1) Mod 3 + 1, (i - 1) \ 3 + 1) = _
            .Fields(0).Value
        .Move Next
    Loop
End With
```

Zwecks besserer Übersicht mit eigenen Variablen für Zeilen- und Spaltenindex:

```
i = 0
With rs
    Do Until .EOF
        i = i + 1
        r = (i - 1) Mod 3 + 1
        c = (i - 1) \ 3 + 1
        Ziel(r, c) = .Fields(0).Value
        .Move Next
    Loop
End With
```

Die vier eingangs erwähnten Varianten würden mit folgenden Code-Varianten erzeugt:

- 3 × 4, N: Mod 3; \ 3
- 4 × 3, N: Mod 4; \ 4
- 3 × 4, Z: \ 3; Mod 3
- 4 × 3, Z: \ 4 Mod 4

Allgemein entstehen

- N-Formen durch Mod - \
- Z-Formen durch \ - Mod.

Als Divisor ist jeweils die erforderliche Zeilenzahl zu verwenden.

Falls die Anzahl der darzustellenden Werte sich nicht so schön aufteilen läßt, wie im Beispiel 12 in 3 × 4 Zellen, bleibt ein Rest von leeren Zellen normalerweise unten rechts in der Ecke. Wenn man festlegt, daß man eine Ausgabe in s Spalten wünscht, und n Werte hat, ist die benötigte Zeilenanzahl (n - 1) \ s + 1 und die Anzahl der Restzellen in der letzten Zeile s - 1 - (n - 1) Mod s

Es wird häufig vorkommen, daß man mehrere Felder eines Datensatzes blockweise ausgeben will, also zum Beispiel:

1	2	3
Eins	Zwei	Drei
4	5	6
Vier	Fünf	Sechs

7	8	9
Sieben	Acht	Neun
10	11	12
Zehn	Elf	Zwölf

Zunächst liegen die Daten in zwölf Datensätzen mit je zwei Feldern „Zahl“ und „Text“ vor. Es werden aber diesmal nicht 4×3 sondern 8×3 (oder 4×6) Matrizenplätze benötigt. Der Schleifenindex reicht immer noch über 12 Nummern. Die doppelte Zeilenzahl erreicht man wie folgt:

```
Dim Ziel(1 To 8, 1 To 3) As String
i = 0
With rs
    Do Until .EOF
        i = i + 1
        r = (i - 1) \ 4 + 1
        c = (i - 1) Mod 4 + 1
        Ziel(2 * r - 1, c) = .Fields(0).Value
        Ziel(2 * r, c) = .Fields(1).Value
        .MoveNext
    Loop
End With
```

Wollte man die Datensatzblöcke nicht untereinander, sondern nebeneinander erzeugen, würde man benutzen:

```
Dim Ziel(1 To 4, 1 To 6) As String
i = 0
With rs
    Do Until .EOF
        i = i + 1
        r = (i - 1) \ 4 + 1
        c = (i - 1) Mod 4 + 1
        Ziel(r, 2 * c - 1) = .Fields(0).Value
        Ziel(r, 2 * c) = .Fields(1).Value
        .Move Next
    Loop
End With
```

Allgemein:

- 2n - 1, 2n
- 3n - 2, 3n - 1, 3n
- 4n - 3, 4n - 2, 4n - 1, 4n

etc.

Natürlich sind auch die Arrays passend zu vergrößern.

Öfter als ein solches Ausgeben einer Datenmenge in einer Rechteckform wird man den Fall benötigen, daß den Spalten eine bestimmte Bedeutung zukommt, während die Zeilen, wenn auch in sortierter Form, die mehr oder weniger zufällig vorhandenen Daten untereinander listen. Solche Spaltenbedeutungen sind gerne Kategorien aller Art, also zum Beispiel Umsatzzahlen nach Datum

untereinander in vier Spalten für 1. bis 4. Quartal oder zwölf Spalten für zwölf Monate. Oft kommen auch sachliche Kategorien vor wie Zahlen spaltenweise nach Konto oder produzierte Einheiten nach Produktionsort oder bearbeitete Vorgänge nach Produktionsgruppenleiter.

	Kategorie 1	Kategorie 2	Kategorie 3	Kategorie 4
Summe	**400**	**800**	**200**	**500**
Anzahl	**7**	**3**	**4**	**2**
Vorgänge	V11	V21	V31	V41
	V12	V22	V32	V42
	V13	V23	V33	
	V14		V34	
	V15			
	V16			
	V17			

Auch hier ergibt sich ein Rechteckschema, das eben über einige leere Plätze verfügt. Im Unterschied zum ersten Beispiel liegt die Spaltenzahl zwar vorher fest, die Zeilenanzahl ergibt sich aber nicht daraus und der Anzahl der Datensätze, da nicht alle Spalten gleich viele Daten enthalten müssen. Vielmehr muß die Array-Tiefe durch die Datensatzanzahl der größten Kategorie bestimmt werden.

Der Rest entspricht dem bereits gezeigten Vorgehen: Frage die Datensätze in geeigneter Sortierung ab und ermittle je eine Regel, die aus einer laufenden Nummer je eine Folge für Zeilen- und Spaltenindizes erzeugt. Ein Beispiel dazu, das direkt je Datensatz drei Ausgabezellen untereinander vorsieht:

```
Dim rcs As DAO.Recordset
Dim db As DAO.Database
Dim SQL As String

Dim Header() As Variant
Dim Ziel() As Variant
Dim Rows As Long, Cols As Long
Dim c1 As Long, c2 As Long
Dim r As Long

Dim i As Long, n As Long
Dim Kat As Long

Set db = CurrentDb
SQL = "SELECT DISTINCT Kategorie FROM tdatBlocktest"
Set rcs = db.OpenRecordset(SQL)
With rcs
    .MoveLast
    Kat = .RecordCount
    ReDim Header(1 To Kat)
    .MoveFirst
    Do Until .EOF
```

```
        i = i + 1
        Header(i) = .Fields(0).Value
        .MoveNext
    Loop
    .Close
End With
```

Hier wird die Anzahl der Kategorien (Spalten in der Ausgabe) bestimmt. Dabei kann ein Array mit den Namen dieser Kategorien als Überschriftenzeile befüllt werden. Man kann das auch in ein Array mit den Daten packen, wodurch sich aber alle Indexnummern entsprechend verschieben. Aus Gründen der Übersichtlichkeit im Code ist es angezeigt, hier lieber z. B. je ein Array für Spaltenköpfe, Zeilenköpfe und Daten zu verwenden. So lange man statt eines Arrays nicht gleich tausend, sondern eben nur drei benutzt, hat man dadurch keine nennenswerte Performance-Einbuße zu erwarten.
Die Tatsache, daß wie üblich unkommentierter Code dieser Art mit aufwendigen Verschiebungen in der Indizierung nach drei Jahren von einem anderen Kollegen garantiert frühestens nach etlichen Stunden verstanden wird, wiegt erheblich schwerer als Zeitverluste, die sich bei drei Arrays höchstens im Nanosekundenbereich bewegen. Zunächst wird die größte Array-Tiefe bestimmt:

```
SQL = _
    "SELECT Max(T.AnzahlZeilen) As MaxAnzahlZeilen " & _
    "FROM " & _
    "(" & _
        "SELECT Count(S.Kategorie) AS AnzahlZeilen " & _
        "FROM tdatBlockTest AS S " & _
        "GROUP BY S.Kategorie " & _
    ") AS T"

Set rcs = db.OpenRecordset(SQL)
Rows = rcs.Fields(0).Value
rcs.Close
```

Dann werden die Nutzdaten in der benötigten Reihenfolge abgefragt.

```
SQL = _
    "Select * From tdatBlocktest " & _
    "ORDER BY Kategorie, id"
Set rcs = db.OpenRecordset(SQL)
rcs.MoveLast
n = rcs.RecordCount
rcs.MoveFirst
```

Da je drei Werte je Datensatz untereinander sollen, wird die dreifache Zeilenanzahl benötigt. Die Spaltenanzahl ist die Anzahl der Kategorien.

```
Cols = Kat
Rows = Rows * 3
ReDim Ziel(1 To Rows, 1 To Cols)
```

Jetzt kommt die Schleife über das Recordset:

```
With rcs
    Do Until .EOF
        If c1 <> c2 Then r = 0
        r = r + 1
```

```
        c1 = .Fields("Kategorie").Value
        Ziel(3 * r - 2, c1) = _
            .Fields(0).Value
        Ziel(3 * r - 1, c1) = _
            .Fields(0).Value & " - " & .Fields(1).Value
        Ziel(3 * r, c1) = _
            .Fields(0).Value & " - " & .Fields(2).Value
        .MoveNext
        If Not .EOF Then c2 = .Fields("Kategorie").Value
    Loop
End With
```

Die eigentliche Übergabe der Daten nach Excel erfolgt dann mit einer einzigen Code-Zeile (die natürlich noch einige Zeilen an Vorbereitung erfordert):

```
Set xlWbk = CreateObject("Excel.Sheet")
Set xlApp = xlWbk.Application
xlApp.Visible = True
Set xlWbk = xlApp.Workbooks.Add

Set xlSht = xlWbk.Worksheets(1)

Set xlRng = xlSht.Range(xlSht.Cells(2, 1), _
                        xlSht.Cells(Rows + 1, Cols))

xlRng.Value = Ziel()

Set xlRng = xlSht.Range(xlSht.Cells(1, 1), _
                        xlSht.Cells(1, Cols))
xlRng.Value = Header()
xlRng.Interior.Color = RGB(196, 196, 196)
xlRng.Font.Bold = True

End Sub
```

Der mathematische Hintergrund dieser Art zu programmieren ist das altbekannte Thema Folgen und Reihen aus der zwölften Klasse.

Es gilt, für die Spaltenindizes und Zeilenindizes je eine Formel zu finden, die diese aus der laufenden Nummer berechnet. Neben der Wiederholung der Grundregeln für arithmetische, geometrische und alternierende Folgen, falls einem diese nicht mehr geläufig sein sollten, ist es das beste Training, gelegentlich einfach mal zu probieren, welche Formeln welche Folgen produzieren oder umgekehrt zu versuchen, zu einfachen Zahlenmustern die Formel aufzustellen.

Sich in seiner Freizeit mit Mathematik zu beschäftigen, ist sowieso das Sinnvollste, was ein Mensch tun kann.

Formelgruppen

Genauso, wie man einen Range mittels eines Arrays in einem Schritt mit Werten befüllt, kann man auch allfällige Formeln in einem Schritt in einen Range schreiben, ohne jede Zelle abklappern zu müssen.

Der lieben Anschauung wegen erst ein Beispiel in deutscher A1-Schreibung: Die Zellen A1 bis A12 mögen irgendwelche Zahlen enthalten. In den benachbarten B-Zellen soll jeweils die laufende Summe von A1 bis zur aktiven Zeile ermittelt werden.

Dazu muß das Blatt folgende Formeln enthalten:

A	B
6	=Summe($A$1:A1)
3	=Summe($A$1:A2)
5	=Summe($A$1:A3)
7	=Summe($A$1:A4)
9	=Summe($A$1:A5)
1	=Summe($A$1:A6)
5	=Summe($A$1:A7)
8	=Summe($A$1:A8)
2	=Summe($A$1:A9)
9	=Summe($A$1:A10)
7	=Summe($A$1:A10)
7	=Summe($A$1:A12)

Das sieht nach Arbeit aus, ist es aber nicht. Der erforderliche Code ist

```
Set xlRng = xlSht.Range("B1:B12")
xlRng.FormulaLocal = "=Summe($A$1:A1)"
```

Die Anpassung A1, A2, A3 wird von Excel automatisch vorgenommen, ohne daß man wie bei Werten dazu ein Array verwenden müßte. Der Trick ist einfach die passende Verwendung der Dollarzeichen:

$A$1 ist immer genau die Zelle A1 (absoluter Bezug). Daher wird $A$1 beim obigen Code auch (wunschgemäß) nicht angepaßt.

A1 ist ein relativer Bezug, d. h. die Zeilennummer paßt sich iterativ nach unten an. Die Spaltennummer würde in einem Range, der auch nach rechts ausgedehnt wäre, dasselbe tun.

$A1 würde nur Zeilen anpassen, Spalte A aber immer beibehalten; A$1 würde nur Spalten anpassen und Zeile 1 immer beibehalten.

Das gleiche als guter Code sähe so aus:

```
Set xlRng = xlSht.Range(xlSht.Cells(1, 2), xlSht.Cells(12, 2))
'Oder
Set xlRng = xlSht.Cells(1, 2).Resize(12, 1)
xlRng.FormulaR1C1 = "=Sum(R1C1:RC[-1])"
```

Das geht ganz analog zur A1-Schreibung. R1C1 ist die Zelle A1 absolut (Row 1, Column 1), R3C2 wäre z. B. B3. Relative Bezüge erkennt Excel an den Klammern; R[0]C[-1] bedeutet also gleiche

Zeile und eine Spalte nach links. Als Kurzschreibweise wird statt R[0] einfach R verwendet, was dann den Teil RC[-1] ergibt.

Wenn man sich den Spaß gönnt, Excel über Extras, Optionen, Allgemein, Z1S1-Bezugsart (oben links auf dem Register) einmal umzustellen, wird man sehen, daß tatsächlich in allen Zellen genau dieselbe Formel steht, nämlich (lokalisiert) =SUMME(Z1S1:ZS(-1))

Deshalb braucht man zum Eintragen der Formeln auch kein Array, es ist nämlich nur eine. Der gewohnte Anblick von A1 bis A12 im Beispiel auf dem Tabellenblatt ist nur Formatierungszauber.

Es gibt allerdings Situationen, in denen man rein aus Faulheit ein Array, das Daten aufnimmt, auch gleich an passenden Stellen mit Formeln für Zwischensummen ausstatten kann. Formeln können beim Schreiben ja auch über Value eingefügt werden.

Zusammengesetzte Bereiche

Man kann mit der Union-Methode des Application-Objekts Bereiche zusammensetzen. Ein solcher Range hat dann nicht nur eine sondern mehrere Areas, die selbst wieder vom Typ Range sind.

```
Set xlRng1 = xlSht.Range("A1:B5")
Set xlRng2 = xlSht.Range("C7:H8")
Set xlRng = xlApp.Union (xlRng1, xlRng2)
Debug.Print xlRng.Areas.Count '> 2
```

xlRng.Areas(1) ist identisch mit xlRng1; xlRng.Areas(2) mit xlRng2.

Man sollte solche Ranges nicht zum Lesen und Schreiben von Daten verwenden – um in den Genuß von Arrays zu kommen, müßte man hierbei sowieso die Areas durchlaufen und könnte dann diese Ranges auch direkt verwenden –, zum Formatieren sind sie aber sehr brauchbar.

Informationen über Ranges

Um wichtige Informationen über ein bestehendes Range-Objekt zu erhalten, kann man folgende Codes verwenden.

Adresse eines Ranges

Die Methode ist nur zum Debuggen interessant, für darauf basierende weitergehende Programmierung sind Zahlenwerte besser, da man sonst den Adreß-String parsen müßte.

```
Debug.Print xlRng.Address
```

Größe eines Ranges

Anzahl Zeilen, Spalten, Zellen

```
Debug.Print xlRng.Rows.Count
```

```
Debug.Print xlRng.Columns.Count
Debug.Print xlRng.Cells.Count
```

Koordinaten eines Ranges

```
'erste Spalte
Debug.Print xlRng.Column

'letzte Spalte, Methode 1
Debug.Print xlRng.Columns(xlRng.Columns.Count).Column

'letzte Spalte, Methode 2
Debug.Print xlRng.Column + xlRng.Columns.Count - 1

'erste Zeile
Debug.Print xlRng.Row

'letzte Zeile, Methode 1
Debug.Print xlRng.Rows(xlRng.Rows.Count).Row

'letzte Zeile, Methode 2
Debug.Print xlRng.Row + xlRng.Rows.Count - 1
```

Row/Column ist eine Zahl; Rows/Columns//Rows(i)/Columns(i) sind Objekte.

Eckzellen als Objekte

```
'Links oben
Set xlRngEdge = xlRng.Cells(1, 1)
'Rechts oben
Set xlRngEdge = xlRng.Cells(1, xlRng.Columns.Count)
'Links unten
Set xlRngEdge = xlRng.Cells(xlRng.Rows.Count, 1)
'Rechts unten
Set xlRngEdge = xlRng.Cells(xlRng.Rows.Count, _
                            xlRng.Columns.Count)
```

Bereiche eines zusammengesetzten Ranges:

```
Debug.Print xlRng Areas.Count
For i = 1 To xlRng.Areas.Count
    Debug.Print xlRng.Area(i).Address
Next i
```

Jede Area ist ein Objekt vom Typ Range und kann entsprechend mit allen einschlägigen Methoden behandelt werden.

Elternobjekte

Tabellenblatt, das den Range enthält:

```
'xlRng.Worksheet
Debug.Print xlRng.Worksheet.Name
```

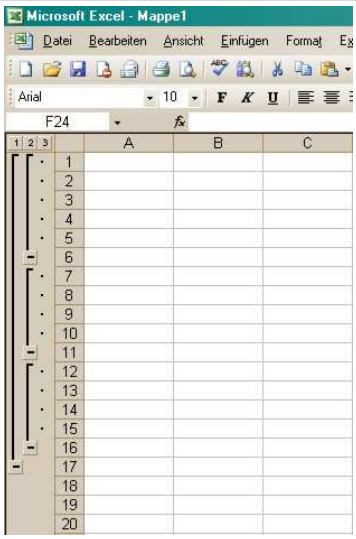
Mappe, die das Tabellenblatt und damit auch den Range enthält:

```
'xlRng.Worksheet.Parent
Debug.Print xlRng.Worksheet.Parent.Name
```

Gliederungen

```
Set xlSht = xlWbk.Worksheets("Tabelle1")

Set xlRng = xlSht.Range("A1:A5")
xlRng.Rows.Group

Set xlRng = xlSht.Range("A7:A10")
xlRng.Rows.Group

Set xlRng = xlSht.Range("A12:A15")
xlRng.Rows.Group

Set xlRng = xlSht.Range("A1:A16")
xlRng.Rows.Group
```

ergibt die Ausgabe wie in der nebenstehenden Grahik:



Zeilen 1 bis 5 werden beim Gliedern ausgeblendet, ebenso 7 bis 10 und 12 bis 15. Die Zeilen 6, 11 und 16 bleiben in der zweiten Ebene sichtbar, z. B. für entsprechende Summenformeln. Die Gliederung über Zeilen 1 bis 16 wird auf eine höhere Ebene plaziert, da sie die anderen Bereiche umfaßt. (Das macht Excel automatisch.) Die Gliederungsebene einer Zeile kann bei vorhandener Gliederung gelesen werden mit:

```
Debug.Print xlSht.Rows(i).OutlineLevel
```

Achtung, die größte Tiefe sind nur sieben Ebenen.

Man kann die Ebene, auf der die Voreinstellung liegen soll, mit

```
xlSht.Outline.ShowLevels 1 ,(2, 3, …)
```

einstellen. Das entspricht einem Klick auf die kleinen Schaltflächen 1 – 2 – 3 oben im Bild.

Interessant ist noch die Eigenschaft

```
xlSht.Outline.SummaryRow = xlSummaryAbove
xlSht.Outline.SummaryRow = xlSummaryBelow
```

mit der festgelegt wird, ob die Aggregatzeile über oder unter dem Gruppenbereich liegt.

Die Syntax für Spaltengruppierungen ist ganz analog.

```
xlRng.Columns.Group
Debug.Print xlSht.Columns(i).OutlineLevel
xlSht.Outline.ShowLevels , 1 'Zweites Argument
xlSht.Outline.SummaryColumn = xlSummaryOnLeft
xlSht.Outline.SummaryColumn = xlSummaryOnRight
```

Formatierungen

Die am häufigsten benötigten Formatierungsbefehle sind folgende:

Schrift

```
xlRng.Font.Bold = True/False
xlRng.Font.Italic = True/False
xlRng.Font.Name = "Schriftartname"
xlRng.Font.Size = 12
```

Rahmen/Schattierung

```
xlrng.Borders(xlEdgeBottom).Color = vbBlack
xlrng.Borders(xlEdgeBottom).LineStyle = xlContinuous
'Linienart und -farbe festlegen

xlrng.Borders(xlEdgeBottom)
xlrng.Borders(xlEdgeLeft)
xlrng.Borders(xlEdgeTop)
xlrng.Borders(xlEdgeRight)
'Unten, links, oben, rechts werden separat eingestellt

xlrng.Borders(xlInsideHorizontal)
xlrng.Borders(xlInsideVertical)
'Innere Rahmenlinien (Gitternetz)

xlRng.Interior.Color = vbRed
'Füllfarbe
```

Textausrichtung

```
With xlRng
    .HorizontalAlignment = xlLeft 'Linksbündig
    .VerticalAlignment = xlBottom 'Unten ausgerichtet
    .WrapText = False             'Textumbruch in Zelle
    .Orientation = 0              'Text drehen
    .AddIndent = False            'Einzug in Zelle
    .IndentLevel = 0              'Wieviel Einzug?
End With
```

Wenn man Textumbruch in einer Zelle erlaubt, ist bei dem String, den man dort per Code einfügen will zu beachten, daß Excel keinen vbCrLf als Zeilenwechsel akzeptiert. Das Zeilenwechselzeichen von Excel ist ein reiner ASCII 12 (*nicht* 13!).

Ein Problem mit allen Farbangaben besteht darin, daß Excel in einem Blatt nur eine Palette von 56 Farben unterstützt.

Wenn man mit der Color-Eigenschaft eine RGB-Farbe angibt, verwendet Excel die dem am nächsten kommende Palettenfarbe, die nicht unbedingt den ästhetischen Vorstellungen des Entwicklers entspricht.

Die Alternative ColorIndex verwendet eine der 56 Farben – diese können in verschiedenen Mappen durch Benutzer aber sonstwie angepaßt worden sein.

Die einzig sichere Methode ist, die Farbpalette des Workbooks per Code festzulegen und dann Farben aus dieser Palette zu verwenden.

```
With xlWbk
    .Colors(1) = RGB(23, 78, 230)
    .Colors(2) = RGB(30, 40, 50)
    …
    .Colors(56) = RGB(200, 180, 252)
End With
```

Man kann nun die gleichen RGB-Farben über Color oder die Indexnummern mit ColorIndex verwenden und hat dann sicher genau die festgelegten Farben.

Relationale Zugriffe auf Excel

Man kann Excel-Tabellen wie gewöhnliche Datenbanktabellen mit SQL abfragen und manipulieren respective mit DAO oder ADO Recordsets darauf aufbauen. Das Andeuten dieser Möglichkeiten bei Datenbankkollegen, die das noch nicht kennen, führt gern zu leuchtenden Augen.

Zu Unrecht! Beim Lesen nach diesen Methoden aus Excel ist vorausgesetzt, daß die Daten dort ebenso relational vorliegen wie in einer Datenbanktabelle. Das kommt gelegentlich vor, ist aber nicht die Regel. Typische Kundenanforderungen sehen eher so aus, daß Excel-Tabellen einzulesen sind, die dermaßen nach Kraut und Rüben aussehen, daß man eigentlich eine KI mit erheblichen hellseherischen Fähigkeiten zu diesem Zwecke programmieren müßte.

Auch eine rein tabellarische Datenausgabe ist zu Präsentationszwecken zumeist nicht gewünscht und auch nicht ergonomisch sinnvoll.

Der Nutzen dieses Zugriffs auf Excel als ISAM-Datenbank ist also höchst beschränkt. Dennoch sollen die verschiedenen Möglichkeiten hier kurz vorgestellt werden.

CopyFromRecordset

Das Allereinfachste zu Beginn:

```
xlRng.CopyFromRecordset rs
```

Als Range genügt eine Zelle, die linke obere Ecke. Das Recordset kann beliebig vom Typ DAO oder ADODB sein. Excel erweitert den Range automatisch auf die nötige Spalten- und Zeilenzahl.

SQL

SQL-Code wie der folgende kann als Abfrage gespeichert und dann ganz normal wie jede andere Abfrage mit DAO oder ADODB behandelt werden.

Ganzes Tabellenblatt:

```
'Eine Zeile
SELECT * FROM [Tabelle1$] IN
 'c:\test\test.xls' [Excel 8.0;HDR=No;IMEX=0;]
'ODER
'Eine Zeile
SELECT * FROM [Excel 8.0;HDR=No;IMEX=0;database=
c:\test\test.xls].[Tabelle1$]
```

Zellbereich auf Tabellenblatt:

```
'Eine Zeile
SELECT * FROM [Tabelle1$B3:D8] IN
 'c:\test\test.xls' [Excel 8.0;HDR=No;IMEX=0;]
'ODER
'Eine Zeile
SELECT * FROM [Excel 8.0;HDR=No;IMEX=0;database=
c:\test\test.xls].[Tabelle1$B3:D8]
```

Die Angabe der Excel-Version muß stimmen. Für Excel 4 lautet sie 4.0 für Excel 5 und 7 (95) 5.0, von 97 bis 2003 einheitlich 8.0

Vorsicht! Diese Datenformatversionsnummer stimmt nicht mit der Programmversionsnummer (z.B. 10, 11, 12) überein.

Mit HDR=No/Yes wird angegeben, daß die erste Zeile des Bereichs Feldnamen enthält. Bei HDR=No lauten die Feldnamen F1, F2, F3 etc. und die erste Zeile des Bereichs wird bereits als Daten interpretiert.

IMEX=0/1/2 gibt in dieser Reihenfolge den Export-, Import- und Verknüpfungsmodus an. Wenn man Daten nach Excel schreiben will ist das ein Export, also IMEX=0. Andere IMEX-Werte lassen keine Bearbeitung zu.

Statt das oben gezeigte SQL als Abfrage zu speichern, kann man es natürlich auch im Code zusammensetzen und als Quelle für ein DAO- oder ADODB-Recordset verwenden.

```
tWbkName = "c:\test\test.xls"
tShtName = "Tabelle1" 'Blattname
tRngName = "B3:D8"    'oder "Bereichsname"

tXLFrom = "[" _
        & tShtName _
        & "$" _
        & tRngName _
        & "]" _
        & "AS Txl IN '" _
        & tWbkName _
        & "'[Excel 8.0;HDR=No;IMEX=0;]"

SQL = "SELECT * FROM " & tXLFrom

'DAO
Set dbd = CurrentDb
Set rsd = dbd.OpenRecordset(SQL)

Do Until rsd.EOF
    For Each fld In rsd.Fields
        Debug.Print fld.Value
```

```
    Next fld
    'rsd.Edit
    'rsd.Update  'ist erlaubt
    'rsd.AddNew  'ist erlaubt
    'rsd.Delete  'ist verboten
    rsd.MoveNext
Loop

rsd.Close

'ADODB
Set cnn = CurrentProject.Connection
Set rsa = New ADODB.Recordset

With rsa
    Set .ActiveConnection = cnn
    .CursorLocation = adUseClient
    .CursorType = adOpenStatic
    .LockType = adLockOptimistic
End With

With rsa
    .Source = SQL
    .Open
    Do Until .EOF
        For Each fld In rsa.Fields
            Debug.Print fld.Value
        Next fld
        'rsa.Update  'ist erlaubt
        'rsa.AddNew  'ist erlaubt
        'rsa.Delete  'ist verboten
        .MoveNext
    Loop
    .Close
End With
```

Statt eines SELECT-Statements kann man natürlich auch UPDATE und INSERT INTO verwenden. DELETE wird allerdings beim ISAM-Zugriff nicht unterstützt.

```
tWbkName = tPath & "Isam.xls"
tShtName = "Quelle"
tRngName = "A1:C8"

'Von Excel nach Access schreiben
SQL = _
    "INSERT INTO tblXLTest (Feld1, Feld2, Feld3) " & _
    "SELECT Feld1, Feld2, Feld3 " & _
    "FROM [Excel 8.0;HDR=Yes;IMEX=1;database=" & _
    tWbkName & "].[" & tShtName & "$" & tRngName & "]"

CurrentDb.Execute SQL
'ODER
CurrentProject.Connection.Execute SQL
'bzw. Ohne Access: cnn.Execute,
'wobei die Connection cnn entsprechend aufgebaut wurde

tShtName = "Ziel"
tRngName = ""
'Von Access nach Excel schreiben
SQL = _
    "INSERT INTO [Excel 8.0;HDR=Yes;IMEX=0;database=" & _
    tWbkName & "].[" & tShtName & "$" & tRngName & _
```

```
    "] (Feld1, Feld2, Feld3) " & _
     "SELECT Feld1, Feld2, Feld3 " & _
     "FROM tblXLTest"

CurrentDb.Execute SQL
'ODER
CurrentProject.Connection.Execute SQL
'bzw. Ohne Access: cnn.Execute,
'wobei die Connection cnn entsprechend aufgebaut wurde
```

Beim Einlesen kann man einen Zellbereich angeben, falls die Daten nicht bündig oben links beginnen. Das **Importieren** von Excel-Daten nach dieser Methode ist problemlos.

Wenn man so nach Excel **exportiert**, sollte man *keinen* Zellbereich angeben. Wenn dieser nicht groß genug ist, um die Daten aufzunehmen, wird sonst ein Fehler ausgelöst. Wenn keine Bereichsangabe erfolgt, wird ab A1 ein beliebig großer Bereich – abhängig von der Datenmenge – verwendet.

Diese zweite Methode des Exports hat jedoch einige höchst gewöhnungsbedürftige Eigenarten. Die Daten werden, wie bei INSERT nicht anders zu erwarten, angefügt. Wenn man das weiß und das erwünscht ist, ist alles in Ordnung. Wenn man aber mit derselben Tabelle immer wieder am ursprünglichen Beginn schreiben möchte, geht das nicht, da hier bereits Datensätze sind. Diese lassen sich nicht mit DELETE löschen.

Wenn man in Excel die Daten durch Markieren und ENTF löscht, ist das meist in Ordnung. Es genügt aber, wenn eine Zelle im Bereich irgendeinen Inhalt *oder auch nur eine Formatierung* enthält, damit Excel annimmt, bis hierhin lägen leere Datensätze vor. Obwohl die Zellen leer sind, gelten sie dann immer noch als Datensätze, deren Felder halt nichts enthalten, so daß neue Daten immer noch unten angefügt werden. Erst wenn man solche „vermurksten“ Bereiche mit dem Befehl *Zellen löschen* entfernt, hat man wirklich die Datensätze und nicht nur die Feldinhalte gelöscht.

Per Code kann man das wie folgt machen:

```
xlRng.ClearContents
'Inhalte löschen; entspricht ENTF-Taste
xlRng.Delete Shift:=xlUp
'Zellen löschen; entspricht Befehl Zellen löschen
```

DAO-RS

Man kann auch ein Recordset direkt auf eine Excel-Tabelle legen.

```
Set dbs = _
    DBEngine.OpenDatabase( _
    Left$(CurrentDb.Name, InStrRev(CurrentDb.Name, "\")) & _
    "Isam.xls", _
    False, False, "Excel 8.0;HDR=Yes;IMEX=0;")

'Ganzes Blatt
Set rcs = dbs.OpenRecordset("Tabelle2$", dbOpenDynaset)
```

```
'Zellbereich
Set rcs = dbs.OpenRecordset("Tabelle2$C7:F17", dbOpenDynaset)

With rcs
    Do Until .EOF
        For Each fld In rcs.Fields
            Debug.Print fld.Value
        Next fld
        .MoveNext
    Loop
    .Close
End With
```

Die so erzeugten Recordsets können ganz normal gehandhabt werden (AddNew, Edit sind möglich), nur mit der Einschränkung, daß eben der ISAM-Zugriff kein Delete zuläßt.

Zu beachten: dbs ist hier *nicht* die CurrentDb, sondern ein Verweis auf die ISAM-Datenbankfunktion der Excel-Datei

ADODB-RS

Genau dasselbe geht auch mit ADODB.

```
Set cnn = New ADODB.Connection
With cnn
    .Provider = "Microsoft.Jet.OLEDB.4.0"
    .Mode = adModeShareDenyNone
    .Properties("Data Source").Value = _
       Left$(CurrentDb.Name, InStrRev(CurrentDb.Name, "\")) & _
       "Isam.xls"   'Pfad zur Excel-Datei
    .Properties("Extended Properties") = _
        "Excel 8.0;HDR=Yes;IMEX=0;"
    .Open
End With

Set rcs = New ADODB.Recordset
With rcs
    Set .ActiveConnection = cnn
    .CursorLocation = adUseClient
    .CursorType = adOpenStatic
    .LockType = adLockOptimistic
End With

'Ganze Tabelle
With rcs
    .Source = "SELECT * FROM [Tabelle1$]"
    .Open
    Do Until .EOF
        For Each fld In rcs.Fields
            Debug.Print fld.Value
        Next fld
        .MoveNext
    Loop
    .Close
End With


'Zellbereich
With rcs
```

```
    .Source = "SELECT * FROM [Tabelle2$C7:F17]"
    .Open
    Do Until .EOF
        For Each fld In rcs.Fields
            Debug.Print fld.Value
        Next fld
        .MoveNext
    Loop
    .Close
End With
```

Auch beim Herstellen einer ADODB.Connection zu Excel bzw. beim Verwenden einer Excel-Datei als DAO.Database gilt wieder, daß keine Delete-Operationen unterstützt werden.

Der Unterschied der beiden Methoden besteht darin, daß im ersten Fall die Verbindungsinformationen im SQL-Code stecken, während im zweiten Fall eine entsprechende Database-Variable (DAO) bzw. Connection-Variable (ADODB) die Verbindungsinformationen enthält.

Der entscheidende Nachteil all dieser Vorgehensweisen ist jedoch, daß man die Daten nur relational, also in Datenbanktabellenform ausgeben kann. Lustiger Schnickschnack mit eingebauten Formeln und pivot-artiger Ausgabe, wie er bei der Automation in den Beispielen gezeigt wird, ist so nicht möglich.

Auch ein Einlesen nichtrelationaler Excel-Daten würde nur Unfug produzieren.

Powerpoint

Zu Powerpoint ein kleines Beispiel, das die grundlegenden Objekte zeigt.

```
Dim ppApp As PowerPoint.Application
Dim ppPrs As PowerPoint.Presentation
Dim ppSld As PowerPoint.Slide
Dim ppShp As PowerPoint.Shape
Dim n As Long
Dim tPrs As String

tPrs = "Pfad\Name.ppt"

Set ppApp = CreateObject("PowerPoint.Application")
'PP läßt sowieso nur eine Instanz zu

ppApp.Visible = msoTrue
'Open nur bei sichtbarer Instanz möglich!!! Sonst LZF

Set ppPrs = ppApp.Presentations.Add
'Neue leere Präsentation

Set ppPrs = ppApp.Presentations.Open(tPrs, , msoTrue)
'Neue Präsentation von Vorlage

Set ppPrs = ppApp.Presentations.Open(tPrs)
```

```
'Präsentation öffnen

Set ppSld = ppPrs.Slides.Add(1, ppLayoutBlank)
ppSld.Name = "MeineFolie"
Set ppShp = _
    ppSld.Shapes.AddTextbox( _
        msoTextOrientationHorizontal, 50, 50, 100, 50)
ppShp.TextFrame.TextRange.Text = "Hallo"

n = ppPrs.Slides.Count
Set ppSld = ppPrs.Slides.Add(n + 1, ppLayoutBlank)
Set ppShp = _
    ppSld.Shapes.AddTextbox( _
        msoTextOrientationHorizontal, 50, 50, 100, 50)
ppShp.TextFrame.TextRange.Text = "Ende"

ppPrs.Slides("MeineFolie").Select

    ppPrs.SlideShowSettings.Run

    ppPrs.SlideShowWindow.View.Exit
```

Von diesem Grundgerüst ausgehend, kann man bei Bedarf tiefer ins Objektgestrüpp vorstoßen. Als kleine Warnung: Die PowerPoint-VBA-Hilfe macht den Eindruck, als habe man bei MS nie ernsthaft damit gerechnet, daß jemand damit ernsthaft programmiert. Wahrscheinlich ist das gar nicht mal so falsch.

Beispielprozeduren

Bei allen hier gelisteten Prozeduren bitte beachten: Der Code ist direkt lauffähig, ggf. haben aber beim Herauskopieren aus Modulen in Word die Zeilenumbrüche gelitten.

Excel-Beispiele

Gruppierter Bericht

Erzeugt aus relationalen Daten einen Pivot-Bericht mit Gruppierung und hierarchisch gegliederten Zwischensummen.

```
Public Sub GruppierterBericht()

Dim rcs As DAO.Recordset
Dim dbs As DAO.Database
Dim SQL As String

Dim Ziel() As Variant
Dim ColHeader(1 To 1, 1 To 5) As String
Dim RowHeader(1 To 17, 1 To 1) As Variant
Dim c As Long
Dim r As Long

Dim xlApp As Excel.Application
```

```
Dim xlWbk As Excel.Workbook
Dim xlSht As Excel.Worksheet
Dim xlRng As Excel.Range

Dim i As Long, n As Long

Dim StartRow As Long, StartCol As Long
StartRow = 2 '5
StartCol = 1 '3

SQL = "SELECT Sum(tdatQuartalGebiet.dtWert) AS dtWert,
tdatQuartalGebiet.dtGebiet, Month([dtDatum]) AS dtMonth " & _
"FROM tdatQuartalGebiet " & _
"GROUP BY tdatQuartalGebiet.dtGebiet, Month([dtDatum]) " & _
"ORDER BY tdatQuartalGebiet.dtGebiet, Month([dtDatum])"

'Ergibt 12 Monatssummen für 4 Gebiete = 48 Datensätze
ReDim Ziel(1 To 17, 1 To 5)
'5 Gebietsspalten; 4 + 1 für Summe
'17 Monatsspalten, 12 + 4 für Quartalssummen + 1 für
Jahressumme

Set dbs = CurrentDb
Set rcs = dbs.OpenRecordset(SQL)
With rcs
    .MoveLast
    n = .RecordCount
    .MoveFirst
    Do Until .EOF
        r = .Fields("dtMonth").Value
        r = r + (r - 1) \ 3
        'Macht aus der Zahlenreihe 1,2,3,4,5,6,7,8,9,10,11,12
        'die Reihe                 1,2,3,5,6,7,9,10,11,13,14,15
        'da                       4,8,12,16
        'für Quartalssummen frei bleiben müssen
        c = .Fields("dtGebiet").Value
        Ziel(r, c) = .Fields("dtWert").Value
        RowHeader(r, 1) = .Fields("dtMonth").Value
        .MoveNext
    Loop
End With

For c = 1 To 4
    For r = 4 To 16 Step 4
        Ziel(r, c) = "=SUM(R[-3]C:R[-1]C)"
        RowHeader(r, 1) = "Q " & CStr(r \ 4)
    Next r
    Ziel(17, c) = "=SUM(R[-13]C,R[-9]C,R[-5]C,R[-1]C)"
Next c
RowHeader(17, 1) = "Jahr"

For r = 1 To 17
    Ziel(r, 5) = "=SUM(RC[-4]:RC[-1])"
Next r

ColHeader(1, 1) = "Nord"
ColHeader(1, 2) = "Süd"
ColHeader(1, 3) = "West"
ColHeader(1, 4) = "Ost"
ColHeader(1, 5) = "DE"


Set xlWbk = CreateObject("Excel.Sheet")
```

```
Set xlApp = xlWbk.Application
xlApp.Visible = True
Set xlWbk = xlApp.Workbooks.Add

Set xlSht = xlWbk.Worksheets("Tabelle1")

Set xlRng = xlSht.Cells(StartRow, StartCol + 1).Resize(1, 5)
xlRng.Value = ColHeader()
xlRng.Font.Bold = True
xlRng.HorizontalAlignment = xlCenter

Set xlRng = xlSht.Cells(StartRow + 1, StartCol).Resize(17, 1)
xlRng.Value = RowHeader()
xlRng.Font.Bold = True
xlRng.HorizontalAlignment = xlRight

Set xlRng = xlSht.Cells(StartRow + 1, StartCol + 1).Resize(17,
5)
xlRng.Value = Ziel()

Set xlRng = xlApp.Union( _
            xlSht.Cells(StartRow + 1 * 4, StartCol +
1).Resize(1, 4), _
            xlSht.Cells(StartRow + 2 * 4, StartCol +
1).Resize(1, 4), _
            xlSht.Cells(StartRow + 3 * 4, StartCol +
1).Resize(1, 4), _
            xlSht.Cells(StartRow + 4 * 4, StartCol +
1).Resize(1, 4), _
            xlSht.Cells(StartRow + 1, StartCol + 5).Resize(16,
1) _
            )
xlRng.Interior.Color = RGB(192, 192, 192)

Set xlRng = xlSht.Cells(StartRow + 4 * 4 + 1, StartCol +
1).Resize(1, 5)
xlRng.Interior.Color = RGB(128, 128, 128)

xlSht.Cells(1, StartCol + 1).Resize(1, 4).Columns.Group
xlSht.Cells(StartRow + 1 + 0 * 4, 1).Resize(3, 1).Rows.Group
xlSht.Cells(StartRow + 1 + 1 * 4, 1).Resize(3, 1).Rows.Group
xlSht.Cells(StartRow + 1 + 2 * 4, 1).Resize(3, 1).Rows.Group
xlSht.Cells(StartRow + 1 + 3 * 4, 1).Resize(3, 1).Rows.Group
xlSht.Cells(StartRow + 1, 1).Resize(4 * 4, 1).Rows.Group


End Sub
```

Periodensystem

Erzeugt aus einer relationalen Elementliste die aus dem Chemieunterricht bekannte Darstellung natürlich in Langform zum Aufklappen. Zwei Varianten. (Jeff Pike hätte wahrscheinlich die entsprechende Tabelle in der Datenbank bereits in dieser Form abgelegt.)

```
Public Function Spalte(oz As Long) As Long

Select Case oz
    Case 57 To 86, 89 To 118
        Spalte = (oz - 55) Mod 32 + 1
```

```
    Case 21 To 36, 39 To 54
        Spalte = (oz - 19) Mod 18 + 15
    Case 5 To 10, 13 To 18
        Spalte = (oz - 3) Mod 8 + 25
    Case 1, 3, 11, 19, 37, 55, 87
        Spalte = 1
    Case 4, 12, 20, 38, 56, 88
        Spalte = 2
    Case 2
        Spalte = 32
End Select

End Function


Public Function Zeile(oz As Long) As Long

Select Case oz
    Case 55 To 118
        Zeile = (oz - 55) \ 32 + 6
    Case 19 To 54
        Zeile = (oz - 19) \ 18 + 4
    Case 3 To 18
        Zeile = (oz - 3) \ 8 + 2
    Case 1, 2
        Zeile = 1
End Select

End Function


Public Function ElKonfig(Spalte As Long) As String

Select Case Spalte
    Case 1, 2
        ElKonfig = "s" & Spalte
    Case 3 To 16
        ElKonfig = "f" & Spalte - 2
    Case 17 To 26
        ElKonfig = "d" & Spalte - 16
    Case 27 To 32
        ElKonfig = "p" & Spalte - 26
End Select

End Function


Public Sub PSAusgabe1()

Dim i As Long, j As Long
Dim rs As DAO.Recordset
Dim db As DAO.Database
Dim ps() As Variant
Dim oz As Long, Col As Long, Row As Long

Dim rStart As Long, cStart As Long
Dim rWidth As Long, cWidth As Long


Dim xlApp As Excel.Application
Dim xlWbk As Excel.Workbook

Dim xlSht As Excel.Worksheet
```

```
Dim xlRng As Excel.Range

rWidth = 21
cWidth = 32

ReDim ps(0 To rWidth, 0 To cWidth)

Set db = CurrentDb
Set rs = db.OpenRecordset("tblElemente")

With rs
    Do Until .EOF
        oz = .Fields("OZ").Value
        Row = Zeile(oz)
        Col = Spalte(oz)
        ps(3 * Row - 2, 0) = Row 'Periodennummern
        ps(0, Col) = ElKonfig(Spalte(oz))
        ps(3 * Row - 2, Col) = oz
        ps(3 * Row - 1, Col) = .Fields("Symbol").Value
        ps(3 * Row, Col) = "" & .Fields("Bezeichnung").Value
        .MoveNext
    Loop
    .Close
End With
Set rs = Nothing
Set db = Nothing

Set xlWbk = CreateObject("Excel.Sheet")
Set xlApp = xlWbk.Application

Set xlWbk = xlApp.Workbooks.Add

Set xlSht = xlWbk.Worksheets("Tabelle1")

rStart = 3
cStart = 2
If rStart > 1 _
    Then
        xlSht.Cells(rStart - 1, cStart).Value = "Periodensystem
der Elemente"
        xlSht.Cells(rStart - 1, cStart).Font.Size = 24
End If

Set xlRng = xlSht.Range(xlSht.Cells(rStart, cStart),
xlSht.Cells(rStart + rWidth, cStart + cWidth))

xlRng.Value = ps    'Das ist die eigentliche Datenübertragung

rStart = rStart - 1
cStart = cStart - 1

xlSht.Cells.Interior.Color = vbWhite

xlSht.Outline.SummaryColumn = xlSummaryOnRight
xlRng.Range(xlRng.Cells(1 - rStart, 4 - cStart), xlRng.Cells(1
- rStart, 17 - cStart)).Columns.Group
xlRng.Range(xlRng.Cells(1 - rStart, 4 - cStart), xlRng.Cells(1
- rStart, 27 - cStart)).Columns.Group

xlRng.Range(xlRng.Cells(1 - rStart, 2 - cStart), xlRng.Cells(22
- rStart, 3 - cStart)).Interior.Color = RGB(255, 128, 128)
xlRng.Range(xlRng.Cells(1 - rStart, 4 - cStart), xlRng.Cells(22
- rStart, 17 - cStart)).Interior.Color = RGB(255, 255, 128)
```

```
xlRng.Range(xlRng.Cells(1 - rStart, 18 - cStart),
xlRng.Cells(22 - rStart, 27 - cStart)).Interior.Color = RGB(0,
255, 0)
xlRng.Range(xlRng.Cells(1 - rStart, 28 - cStart),
xlRng.Cells(22 - rStart, 33 - cStart)).Interior.Color =
RGB(128, 128, 255)

With xlRng.Range(xlRng.Cells(1 - rStart, 1 - cStart),
xlRng.Cells(1 - rStart, 33 - cStart)).Borders(xlEdgeBottom)
    .Color = vbBlack
    .LineStyle = xlSolid
    .Weight = xlMedium
End With

For i = 4 To 22 Step 3
With xlRng.Range(xlRng.Cells(i - rStart, 1 - cStart),
xlRng.Cells(i - rStart, 33 - cStart)).Borders(xlEdgeBottom)
    .Color = vbBlack
    .LineStyle = xlSolid
    .Weight = xlThin
End With

Next i

xlRng.Range(xlRng.Cells(1 - rStart, 1 - cStart), xlRng.Cells(22
- rStart, 1 - cStart)).Font.Bold = True
xlRng.Range(xlRng.Cells(1 - rStart, 1 - cStart), xlRng.Cells(1
- rStart, 33 - cStart)).Font.Bold = True

xlRng.HorizontalAlignment = xlLeft
xlRng.ColumnWidth = 10
xlRng.Cells(1, 1).ColumnWidth = 2
xlSht.Outline.ShowLevels , 2
xlSht.Outline.ShowLevels , 1

xlApp.Visible = True

End Sub

Public Sub PSAusgabe2()

Dim i As Long, j As Long
Dim rs As DAO.Recordset
Dim db As DAO.Database
Dim ps() As Variant
Dim oz As Long, Col As Long, Row As Long

Dim rStart As Long, cStart As Long
Dim rWidth As Long, cWidth As Long


Dim xlApp As Excel.Application
Dim xlWbk As Excel.Workbook

Dim xlSht As Excel.Worksheet
Dim xlRng As Excel.Range

rWidth = 14
cWidth = 32

ReDim ps(0 To rWidth, 0 To cWidth)

Set db = CurrentDb
```

```
Set rs = db.OpenRecordset("tblElemente")

With rs
    Do Until .EOF
        oz = .Fields("OZ").Value
        Row = Zeile(oz)
        Col = Spalte(oz)
        ps(2 * Row - 1, 0) = Row 'Periodennummern
        ps(0, Col) = ElKonfig(Spalte(oz))
        ps(2 * Row - 1, Col) = oz
        ps(2 * Row, Col) = .Fields("Symbol").Value
        .MoveNext
    Loop
    .Close
End With
Set rs = Nothing
Set db = Nothing



Set xlWbk = CreateObject("Excel.Sheet")
Set xlApp = xlWbk.Application

Set xlWbk = xlApp.Workbooks.Add

Set xlSht = xlWbk.Worksheets("Tabelle1")

rStart = 3
cStart = 2
If rStart > 1 _
    Then
        xlSht.Cells(rStart - 1, cStart).Value = "Periodensystem
der Elemente"
        xlSht.Cells(rStart - 1, cStart).Font.Size = 24
End If



Set xlRng = xlSht.Range(xlSht.Cells(rStart, cStart),
xlSht.Cells(rStart + rWidth, cStart + cWidth))

xlRng.Value = ps    'Das ist die eigentliche Datenübertragung

rStart = rStart - 1
cStart = cStart - 1

xlSht.Cells.Interior.Color = vbWhite

xlSht.Outline.SummaryColumn = xlSummaryOnRight
xlRng.Range(xlRng.Cells(1 - rStart, 4 - cStart), xlRng.Cells(1
- rStart, 17 - cStart)).Columns.Group
xlRng.Range(xlRng.Cells(1 - rStart, 4 - cStart), xlRng.Cells(1
- rStart, 27 - cStart)).Columns.Group

xlRng.Range(xlRng.Cells(1 - rStart, 2 - cStart), xlRng.Cells(15
- rStart, 3 - cStart)).Interior.Color = RGB(255, 128, 128)
xlRng.Range(xlRng.Cells(1 - rStart, 4 - cStart), xlRng.Cells(15
- rStart, 17 - cStart)).Interior.Color = RGB(255, 255, 128)
xlRng.Range(xlRng.Cells(1 - rStart, 18 - cStart),
xlRng.Cells(15 - rStart, 27 - cStart)).Interior.Color = RGB(0,
255, 0)
```

```
xlRng.Range(xlRng.Cells(1 - rStart, 28 - cStart),
xlRng.Cells(15 - rStart, 33 - cStart)).Interior.Color =
RGB(128, 128, 255)

With xlRng.Range(xlRng.Cells(1 - rStart, 1 - cStart),
xlRng.Cells(1 - rStart, 33 - cStart)).Borders(xlEdgeBottom)
    .Color = vbBlack
    .LineStyle = xlSolid
    .Weight = xlMedium
End With

For i = rStart + 1 To rStart + rWidth Step 2
With xlRng.Range(xlRng.Cells(i - rStart, 1 - cStart),
xlRng.Cells(i - rStart, 33 - cStart)).Borders(xlEdgeBottom)
    .Color = vbBlack
    .LineStyle = xlSolid
    .Weight = xlThin
End With

Next i

xlRng.Range(xlRng.Cells(1 - rStart, 1 - cStart), xlRng.Cells(15
- rStart, 1 - cStart)).Font.Bold = True
xlRng.Range(xlRng.Cells(1 - rStart, 1 - cStart), xlRng.Cells(1
- rStart, 33 - cStart)).Font.Bold = True

xlRng.ColumnWidth = 4
xlRng.Cells(1, 1).ColumnWidth = 2
xlSht.Outline.ShowLevels , 3

xlApp.Visible = True

End Sub
```

Auswertung einer poisson-verteilten Probe

Erzeugt aus einer kumulierten Probe die zugehörigen Poisson-Wahrscheinlichkeiten.

```
Public Sub PoissonTest()

Dim rcs As DAO.Recordset
Dim dbs As DAO.Database
Dim SQL As String

Dim Ziel() As Variant
Dim r As Long

Dim xlApp As Excel.Application
Dim xlWbk As Excel.Workbook
Dim xlSht As Excel.Worksheet
Dim xlRng As Excel.Range

Dim i As Long, n As Long

SQL = _
    "SELECT X, Hx " & _
    "FROM tdatPoisson " & _
    "ORDER BY X"


Set dbs = CurrentDb
```

```
Set rcs = dbs.OpenRecordset(SQL)
With rcs
    .MoveLast
    n = .RecordCount
    .MoveFirst
    ReDim Ziel(1 To n, 1 To 2)
    Do Until .EOF
        r = r + 1
        Ziel(r, 1) = .Fields("X").Value
        Ziel(r, 2) = .Fields("Hx").Value
        .MoveNext
    Loop
End With

Set xlWbk = CreateObject("Excel.Sheet")
Set xlApp = xlWbk.Application
xlApp.Visible = True
Set xlWbk = xlApp.Workbooks.Add

Set xlSht = xlWbk.Worksheets("Tabelle1")

Set xlRng = xlSht.Cells(3, 1).Resize(n, 2)
xlRng.Value = Ziel()

Set xlRng = xlSht.Cells(n + 3, 2).Resize(1, 2)
xlRng.FormulaR1C1 = "=SUM(R[-" & n & "]C:R[-1]C)"

Set xlRng = xlSht.Cells(3, 3).Resize(n, 1)
xlRng.FormulaR1C1 = "=RC[-2]*RC[-1]"

Set xlRng = xlSht.Cells(3, 4).Resize(n, 1)
xlRng.FormulaR1C1 = "=RC[-2]/R" & n + 3 & "C2"

Set xlRng = xlSht.Cells(3, 5).Resize(n, 1)
xlRng.FormulaR1C1 = "=SUM(R3C4:RC[-1])"

Set xlRng = xlSht.Cells(3, 6).Resize(n, 1)
xlRng.FormulaR1C1 = "=(R" & n + 3 & "C3/R" & n + 3 & "C2)^RC[-
5]/FACT(RC[-5])*EXP(-(R" & n + 3 & "C3/R" & n + 3 & "C2))"

Set xlRng = xlSht.Cells(3, 7).Resize(n, 1)
xlRng.FormulaR1C1 = "=SUM(R3C6:RC[-1])"

Set xlRng = xlSht.Cells(3, 8).Resize(n, 1)
xlRng.FormulaR1C1 = "=ABS(RC[-3]-RC[-1])"

xlSht.Cells(2, 1).Value = "Anzahl Keime pro Probe"
xlSht.Cells(2, 2).Value = "Anzahl Proben mit x Keimen"
xlSht.Cells(2, 3).Value = "Anzahl Keime gesamt"
xlSht.Cells(2, 4).Value = "Relative Häufigkeit(Anteil)"
xlSht.Cells(2, 5).Value = "Kumulierte Häufigkeit"
xlSht.Cells(2, 6).Value = "Poisson-Wk"
xlSht.Cells(2, 7).Value = "Kumulierte Poisson-Wk"
xlSht.Cells(2, 8).Value = "Abweichung"

End Sub
```

Daten nach Kategorie nebeneinander

Erzeugt einen Bericht, der Daten nach Kategorien in Spalten nebeneinanderstellt. Je Datensatz werden drei Felder in einem Block untereinander dargestellt.

```
Public Sub MehrereFelderUntereinanderNachKategorieSpalten()
Dim rcs As DAO.Recordset
Dim db As DAO.Database
Dim SQL As String

Dim Header() As Variant
Dim Ziel() As Variant
Dim Rows As Long, Cols As Long
Dim c1 As Long, c2 As Long
Dim r As Long

Dim xlApp As Excel.Application
Dim xlWbk As Excel.Workbook
Dim xlSht As Excel.Worksheet
Dim xlRng As Excel.Range

Dim i As Long, n As Long

Dim Kat As Long

Set db = CurrentDb

SQL = "SELECT DISTINCT Kategorie FROM tdatBlocktest"
Set rcs = db.OpenRecordset(SQL)
With rcs
    .MoveLast
    Kat = .RecordCount
    ReDim Header(1 To Kat)
    .MoveFirst
    Do Until .EOF
        i = i + 1
        Header(i) = .Fields(0).Value
        .MoveNext
    Loop
    .Close
End With

SQL = _
    "SELECT Max(T.AnzahlZeilen) As MaxAnzahlZeilen " & _
    "FROM " & _
    "(" & _
        "SELECT Count(S.Kategorie) AS AnzahlZeilen " & _
        "FROM tdatBlockTest AS S " & _
        "GROUP BY S.Kategorie " & _
    ") AS T"


Set rcs = db.OpenRecordset(SQL)
Rows = rcs.Fields(0).Value
rcs.Close

SQL = "Select * From tdatBlocktest ORDER BY Kategorie, id"
Set rcs = db.OpenRecordset(SQL)
rcs.MoveLast
n = rcs.RecordCount
rcs.MoveFirst
```

```
'Vorgabe: Kat Spalten
'Da je drei Werte je Datensatz untereinander sollen,
'dreifache Zeilenanzahl (+1 für Spaltenköpfe)
Cols = Kat
Rows = Rows * 3
ReDim Ziel(1 To Rows, 1 To Cols)

With rcs
    Do Until .EOF
        If c1 <> c2 Then r = 0
        r = r + 1
        c1 = .Fields("Kategorie").Value
        Ziel(3 * r - 2, c1) = .Fields(0).Value

        Ziel(3 * r - 1, c1) = .Fields(0).Value & " - " &
.Fields(1).Value

        Ziel(3 * r, c1) = .Fields(0).Value & " - " &
.Fields(2).Value
        .MoveNext
        If Not .EOF Then c2 = .Fields("Kategorie").Value
    Loop
End With

Set xlWbk = CreateObject("Excel.Sheet")
Set xlApp = xlWbk.Application
xlApp.Visible = True
Set xlWbk = xlApp.Workbooks.Add

Set xlSht = xlWbk.Worksheets("Tabelle1")

Set xlRng = xlSht.Range(xlSht.Cells(2, 1), xlSht.Cells(Rows +
1, Cols))
xlRng.Value = Ziel()

Set xlRng = xlSht.Range(xlSht.Cells(1, 1), xlSht.Cells(1,
Cols))
xlRng.Value = Header()
xlRng.Interior.Color = RGB(196, 196, 196)
xlRng.Font.Bold = True
End Sub
```

CopyFromRecordset-Demo

```
Public Sub CopyFromRS()
Dim xlApp As Excel.Application
Dim xlWbk As Excel.Workbook
Dim xlSht As Excel.Worksheet
Dim xlRng As Excel.Range

Dim rsd As DAO.Recordset
Dim dbd As DAO.Database

Dim rsa As ADODB.Recordset
Dim cna As ADODB.Connection

Set xlWbk = CreateObject("Excel.Sheet")
Set xlApp = xlWbk.Application
```

```
xlApp.Visible = True
Set xlWbk = xlApp.Workbooks.Add

'DAO
Set dbd = CurrentDb
Set rsd = dbd.OpenRecordset("SELECT * FROM tdatQuartalGebiet")

Set xlSht = xlWbk.Worksheets("Tabelle1")
Set xlRng = xlSht.Range("C5")

xlRng.CopyFromRecordset rsd

rsd.Close

'ADODB
Set cna = CurrentProject.Connection
Set rsa = New ADODB.Recordset
Set rsa.ActiveConnection = cna
rsa.LockType = adLockOptimistic
rsa.CursorLocation = adUseClient
rsa.Source = "SELECT * FROM tdatQuartalGebiet"
rsa.Open

Set xlSht = xlWbk.Worksheets("Tabelle2")
Set xlRng = xlSht.Range("C5")

xlRng.CopyFromRecordset rsa

rsa.Close

End Sub
```