Einführung in Visual Basic for Applications

Fachgebiet Operations Research – Philipp Schade

29. Dezember 2000

Inhaltsverzeichnis

1. Microsoft Excel – Basiswissen

Bevor wir uns detailliert mit Visual Basic for Applications (VBA) unter Excel auseinandersetzen, sollen kurz grundlegende Elemente der Arbeitsumgebung vorgestellt werden.

1.1. Die Arbeitsumgebung

Die wesentlichen Elemente der Arbeitsoberfläche von Excel sind die Titelzeile, die Menüleiste, die Symbolleiste, die Funktionsleiste, der Arbeitsbereich und schließlich die Statuszeile.

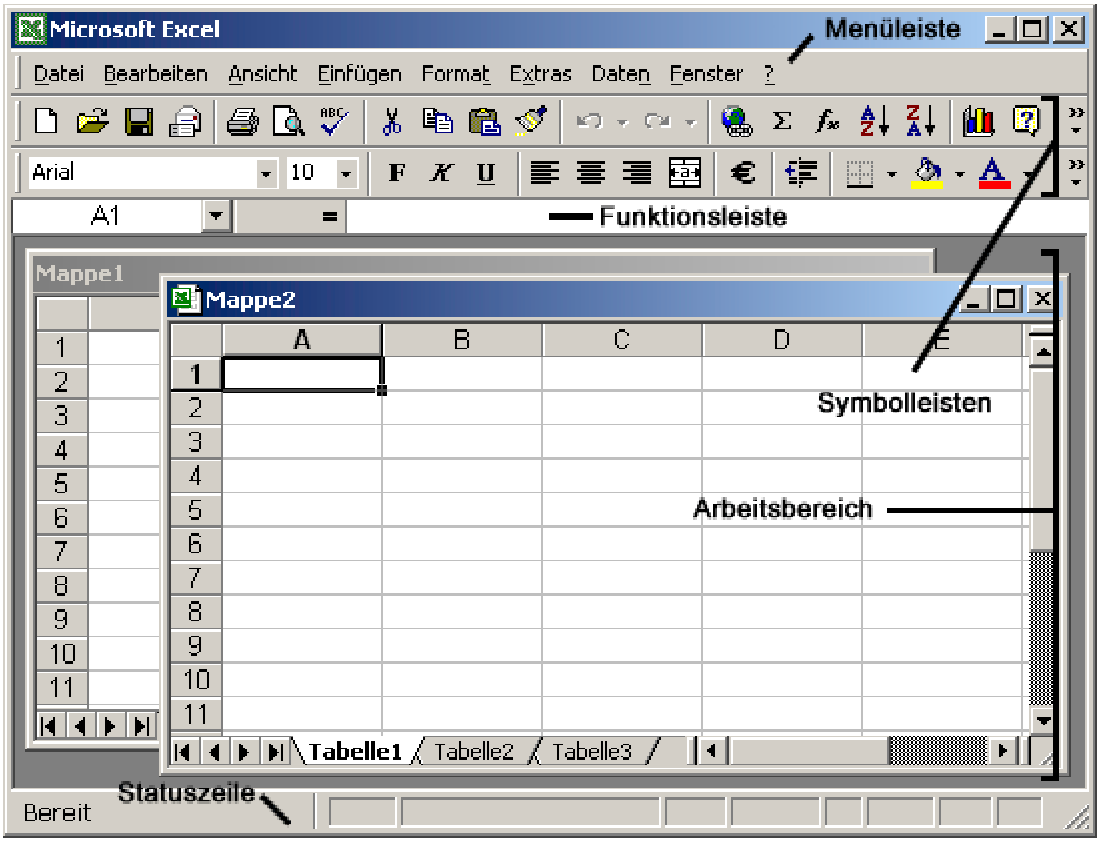

Abbildung 1: Oberfläche von Microsoft Excel

Die Titelzeile der Anwendung enthält immer den Anwendungsnamen und im Fall geöffneter Arbeitsmappen, den Dateinamen der Arbeitsmappe. Die Menüleiste bietet Zugang zu sämtlichen Funktionen, die auf eine Arbeitsmappe anwendbar sind. Da Excel mit variablen Menüs arbeitet, sind die verfügbaren Befehle abhängig vom Inhalt der geöffneten Arbeitsmappe. Die Statuszeile am unteren Rand der Anwendung zeigt Meldungen zum aktuellen Status der Anwendung. Die restlichen wichtigen Elemente der Oberfläche werden im Folgenden vorgestellt.

1.1.1. Arbeitsmappe

Der Arbeitsbereich, der von Excel zur Verfügung gestellt wird, ist einem gewöhnlichen Schreibtischarbeitsplatz nachempfunden. Zu erledigende Arbeitsvorgänge werden in Arbeitsmappen

abgelegt. Arbeitsmappen fassen verschiedene Arbeitsblätter zu einem Gesamtdokument zusammen. Eine solche Mappe enthält demnach inhaltlich zusammenhängende Tabellen, Diagramme, Grafiken und Formulare; darüber hinaus auch bestimmte Arbeitsanweisungen bzw. Funktionshilfen (in unserem Fall sog. Makros) zur Unterstützung der Arbeit. Eine Arbeitsmappe wird immer als separates Fenster innerhalb der Arbeitsumgebung angezeigt.

Microsoft Excel erlaubt in einer Arbeitsmappe (engl. *workbook*) bis zu 256 verschiedene Arbeitsblätter (engl. *worksheet*), mindestens jedoch eins. Ein Blatt der Mappe ist jeweils das aktive Arbeitsblatt, in dem Daten eingegeben oder bearbeitet werden können.

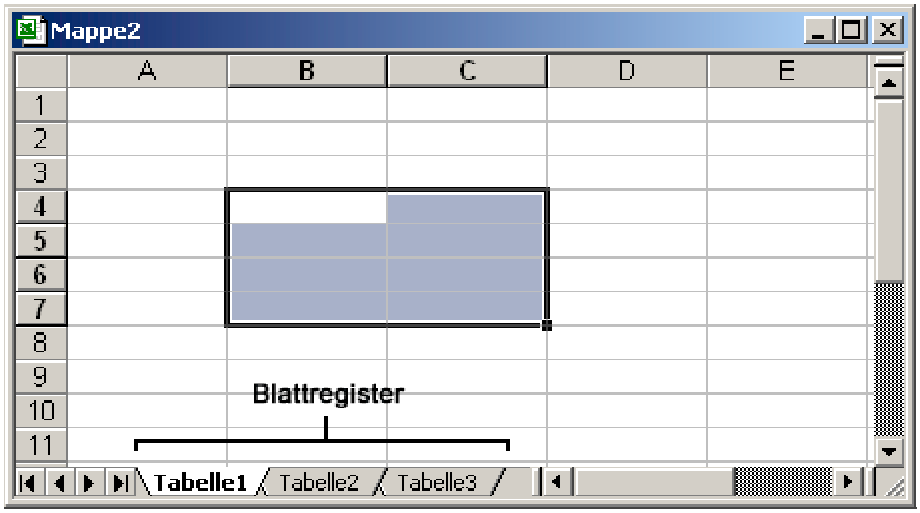

Abbildung 2: Bereich innerhalb eines Tabellenblattes

Jedes Arbeitsblatt innerhalb einer Arbeitsmappe wird über Blattregister am unteren Rand der Mappe angesprochen. Das Blattregister enthält die eindeutigen Namen der Arbeitsblätter. Das aktive Blatt ist jeweils hervorgehoben; der Blattname erscheint in Fettschrift. Mit Hilfe der vier links neben dem Blattregister befindlichen Schaltflächen kann zwischen den Blättern der Mappe hin und her gewechselt werden.

Beim Anlegen einer neuen Arbeitsmappe werden von Excel standardmäßig drei Arbeitsblätter (Tabellenblätter) erzeugt.

Zum Speichern einer Arbeitsmappe verwendet Excel die Dateinamenswerweiterung *'.xls'*. Dabei wird immer die gesamte Arbeitsmappe gespeichert mit allen dazugehörigen Arbeitsblättern. Darüber hinaus können Vorlagen (engl. *templates*) für bestimmte, immer wiederkehrende Aufgaben angelegt werden, die Excel unter der der Dateinamenserweiterung *'.xlt'* ablegt.

1.1.2. Tabellenblatt

Als Arbeitsblätter kommen innerhalb einer Arbeitsmappe grundlegend Tabellen- und Diagrammblätter zum Einsatz. Die größte Bedeutung besitzt das Tabellenblatt. Ähnlich einer Matrix besitzt ein Excel-Tabellenblatt Zeilen und Spalten. Die 256 möglichen Spalten werden mit A bis IV durchnummeriert, die Zeilen entsprechend den Zeilennummern von 1 bis 65536. Jedem Arbeitsblatt muss ein eindeutiger, möglichst sprechender Name zugewiesen werden.

1.1.3. Zelle

Schnittpunkt einer Zeile und Spalte eines Tabellenblattes ist immer eine *Zelle*. Die Zelle ist der Träger der Daten. Jede Zelle besitzt ähnlich einem Schachbrett eine eindeutige Position innerhalb des Tabellenblattes. Man spricht von *Adresse* der Zelle. Die vollständige Adresse einer Zelle lautet z. B.

```
[MeineMappe]Kostentab!F6
```

Im Beispiel wird also die Zelle in Spalte *F*, Zeile *6* des Tabellenblattes *'Kostentab'* in der Arbeitsmappe mit Namen *'MeineMappe'* angesprochen.

Innerhalb eines Tabellenblattes ist immer ein Zelle hervorgehoben und damit markiert. Man nennt diese Zelle die aktive Zelle. Eine aktive Zelle ist bereit für die Aufnahme von Daten. In der unteren rechten Ecke der aktiven Zelle ist immer ein spezielles Kästchen zu sehen – das sog. Ausfüllkästchen. Das Ausfüllkästchen bietet zusätzliche Funktionalität, die bestimmte Arbeitsschritte wesentlich vereinfachen (z. B. das Erzeugen von Datenreihen).

Bei jeder Zelle unterscheidet man zwei Dimensionen, den Inhalt einer Zelle und dessen Format. Beide Dimensionen werden von Excel getrennt verwaltet. Beim Inhalt einer Zelle unterscheidet Excel zwischen zwei Typen von Daten, die eingegeben werden können – Konstanten und Formeln. Konstanten sind numerische Werte bzw. Zeichenfolgen. Formeln sind Rechenvorschriften oder Vorschriften zur Manipulation von Daten der Zelle. Das Ergebnis der Formel wird durch Excel in der Zelle angezeigt.

Darüber hinaus ist es möglich, einer Zelle eine dritte Dimension in Form eines zellbezogenen Kommentars zuzuordnen.

Neben der einzelnen Zelle innerhalb eines Tabellenblattes spielen Zellbereiche eine sehr große Rolle, da Excel Funktionen/Befehle anbietet, die auf mehrere Zellen gleichzeitig angewandt werden können. Mit Bereichen wird eine rechteckige bzw. quaderförmige Gruppe (Auswahl) von Zellen bezeichnet. Der kleinstmögliche Bereich ist die Zelle selber, der größtmögliche Bereich ist die Auswahl aller Zellen eines Tabellenblattes. Darüber hinaus erlaubt Excel auch sog. Mehrfachbereiche, also die Auswahl von mehreren Bereichen gleichzeitig.

Bereichen werden über die Adressen der oberen linken und der unteren rechten Zelle der rechteckigen Auswahl definiert, derjenigen linken und rechten Zelle, die den Bereich aufspannen.

Der in der Abbildung 2 aufgespannte Bereich hat demzufolge die Adresse `B4:C7`. Neben dieser Form der Adressierung können Bereichen auch Namen zugeordnet werden. Auf diese Bereiche wird dann anstatt mit der Bereichsadressierung `Zelle_11:Zelle_nm` mit Hilfe des festgelegten Namens zugegriffen. (Siehe dazu Menü Einfügen|Name|Definieren. . .)

1.1.4. Eingebettete Diagramme und Diagrammblätter

Diagramme sind grafische Darstellungen von Daten, die in Form von Tabellen vorliegen. Werden Diagramme direkt in ein Tabellenblatt eingefügt werden, so spricht man von *eingebetteten Diagrammen*. Darüber hinaus ist es in Excel auch möglich, Diagramme als separates Arbeitsblatt darzustellen. In beiden genannten Fällen sind die Diagramme mit Tabellendaten vorhandener Tabellenblätter verknüpft. Werden die Daten der Tabelle geändert, so passt Excel die Diagramme entsprechend an.

1.1.5. Die Funktionsleiste

Die Funktionsleiste ist wesentlicher Bestandteil der Dateneingabe und -Manipulation innerhalb eines Tabellenblattes. Sie besteht aus drei Teilen. Im ersten Auswahlfeld wird immer die Adresse der aktiven Auswahl angezeigt bzw. bei benannten Bereichen der Name, sobald alle Zellen eines solchen Bereichs markiert sind. Die Schaltfläche mit dem Pfeil nach unten öffnet eine Liste aller vergebenen Namen innerhalb des Tabellenblattes.

Rechts daneben befinden sich die Symboltasten für Eingeben und Abbrechen der Eingabe von Zellinhalten und daneben ein Eingabefeld, das den Inhalt für die aktive Zelle vom Benutzer entgegen nimmt bzw. bei schon vorhandenem Zellinhalt diesen darstellt. Ist der Inhalt der aktiven Zelle eine Formel, so wird im Eingabefeld die Formel, in der Zelle dagegen das Ergebnis dargestellt. Die Symboltasten erscheinen erst, sobald der Benutzer eine Zelle in den Editieren-Status versetzt (Doppelklick).

1.1.6. Die Symbolleiste

Um den Umgang mit oft genutzten Funktionen und Befehlen zu vereinfachen und zu beschleunigen, bietet Excel eine große Auswahl von Symbolschaltflächen an, die in Symbolleisten gruppiert sind. Die Sammlung dieser Symbolleisten wird unterhalb der Menüleiste angezeigt.

Je nach Arbeitsaufgabe werden bestimmte Symbolleisten automatisch ein- und ausgeblendet. Über ein sog. Popup-Menü (Klick mit der rechten Maustaste) kann der Benutzer die Auswahl der Symbolleisten selbst bestimmen und deren Darstellung beeinflussen.

1.2. Die Entwicklungsumgebung – Der Visual Basic-Editor

In diesem Abschnitt wird das Herzstück der Visual Basic-Programmierung unter Excel näher betrachtet. Die Zentrale zur Verwaltung jeglichen Visual Basic-Codes bildet die Entwicklungsumgebung von Excel – der sog. Visual Basic-Editor. Die primäre Aufgabe der Entwicklungsumgebung ist, die Eingabe von Programmcode und die Definition evtl. Formulare zu ermöglichen. In den Versionen bis Excel 7.0 war die Entwicklungsumgebung fester Bestandteil der Anwendungsoberfläche, seit Excel 97 steht sie dagegen als fast eigenständiges Programm da. Zu Erreichen ist die Entwicklungsumgebung über das Menü Extras|Makro|Visual Basic-Editor bzw. durch Drücken der Tastenkombination Alt-F11.

Die Entwicklungsumgebung besteht aus den Komponenten Projektfenster, dem Eigenschaftsfenster, dem Objektkatalog, sowie verschiedenen sog. Modulblättern zur Aufnahme des Programmcodes.

1.2.1. Das Projektfenster

Für jede geladene Arbeitsmappe wird im Projektfenster der Entwicklungsumgebung ein separater Abschnitt erzeugt, der alle Bestandteile der Mappe auflistet, die per Visual Basic-Code erweitert bzw. angepasst werden können. Um das Verhalten der einzelnen Komponenten durch Programmcode verändern oder neue Funktionalitäten hinzufügen zu können, werden Modulblätter durch Doppelklick auf den jeweiligen Eintrag erzeugt bzw. angezeigt. Nur in diesen Blättern kann Visual Basic-Code eingegeben werden. Beim erstmaligen Öffnen der Entwicklungsumgebung einer neu erstellten Arbeitsmappe befinden sich zu jedem Arbeitsblatt und zur Arbeitsmappe bereits vorgefertigte noch leere Module (siehe Abb. 3). In diesen speziellen Modulblättern hat der Benutzer die Möglichkeit, auf bestimmte von Excel bereitgestellte

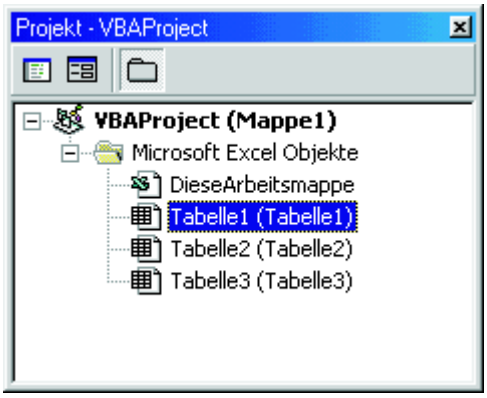

Abbildung 3: Das Projektfenster

Ereignisse (z. B. das Öffnen eines weiteren Arbeitsblattes innerhalb der Mappe) durch sog. Ereignisbehandlungsroutinen zur reagieren. Dazu kann im jeweiligen Modulfenster eine Liste aller verfügbaren Ereignisse angezeigt werden. Um auf ein Ereignis entsprechend zu reagieren, muss dazu geeigneter Programmcode geschrieben werden.

1.2.2. Das Eigenschaftsfenster

Das Eigenschaftsfenster dient zur komfortablen Anpassung von Eigenschaften bestimmter Objekte. Objekte sind dabei entweder Module oder erzeugte Formulare mit sämtlichen in ihnen enthaltenen Steuerelementen. Die einzige Eigenschaft, die bei normalen Modulen angepasst werden kann, ist der Name. Bedeutung gewinnt das Eigenschaftsfenster daher erst richtig bei der Erzeugung von Formularen.

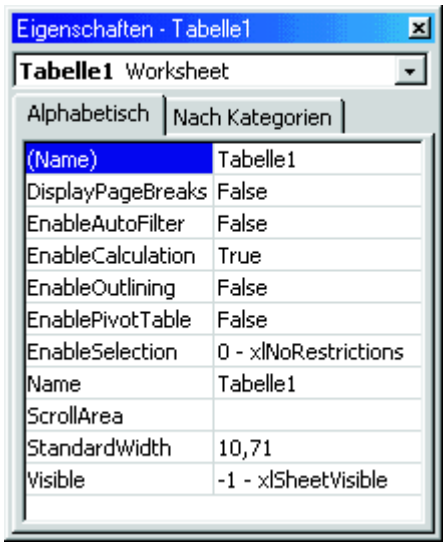

Abbildung 4: Das Eigenschaftsfenster

1.2.3. Module zur Codeeingabe

Wie schon erwähnt, dienen Module oder Modulblätter innerhalb der Entwicklungsumgebung als Container für Programmcode. Standardmäßig besitzt eine neue Arbeitsmappe noch kein Modulblatt. Sollen mit Hilfe von Visual Basic neue Prozeduren und Funktionen geschrieben und der Arbeitsmappe zur Verfügung gestellt werden, so muss erst ein Modul erzeugt werden. Das geschieht in der Entwicklungsumgebung durch den Menübefehl EINFÜGEN|MODUL. Innerhalb einer Arbeitsmappe können auch mehrere Module erzeugt werden, um zum Beispiel bei großen Projekten den Programmcode nach funktionalen oder inhaltlichen Gesichtspunkten zu zerlegen. Ein Modul in Visual Basic ist demzufolge mit einer Unit in Pascal vergleichbar.

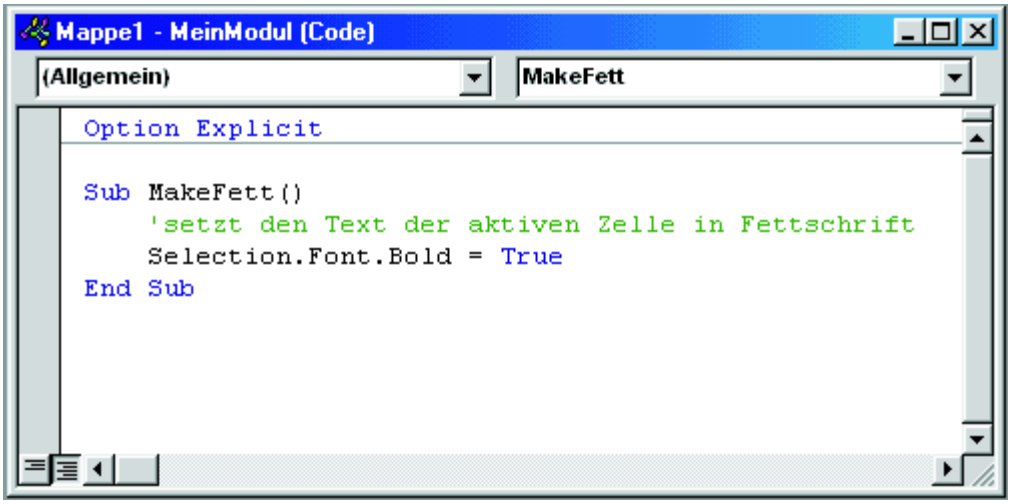

Abbildung 5: Ein Modulblatt mit Beispielcode

Module verfügen gegenüber einem normalen Editor über besondere Erweiterungen. Neben der automatischen Vervollständigung von Programmcode, Schlüsselwortern und Variablennamen verfügen Modulblätter auch über Syntaxhervorhebung und automatischer Syntaxkontrolle. Geschriebener Programmcode wird also nur akzeptiert, wenn er syntaktisch korrekt ist.

1.2.4. Der Objektkatalog

Neben Modulen und deren zentrale Bedeutung soll zum Schluss dieses Abschnitts noch auf den Objektkatalog eingegangen werden.

Die objektorientierte Programmierung mit Visual Basic unter Excel greift auf einer Reihe von vordefinierten Objektbibliotheken zurück. Die wichtigste dieser Bibliotheken ist die Excel-Bibliothek, die alle in Excel verfügbaren Objekte definiert. Wie später noch gezeigt wird, sind Objekte eng mit bestimmten Eigenschaften, Ereignissen und sog. Methoden verknüpft. Aufgrund der Vielzahl ist es nur noch im Objektkatalog möglich, eine übersichtliche Darstellung aller Objekte zu liefern. Hier werden neben den Objekten aller aktivierten Bibliotheken auch die in den Modulen selbst geschriebenen Prozeduren und Funktionen sowie die in sog. Klassenmodulen selbst definierten Objekte aufgelistet. Der Objektkatalog bietet darüber hinaus in vielen Situationen den direkten Weg in die Online-Hilfe.

1.3. Mit dem Latein am Ende – das Hilfesystem

Neben der reinen Online-Hilfe zum Umgang mit der Anwendungsumgebung liefert Microsoft die Visual Basic-Hilfe. Diese ist nur aus der Entwicklungsumgebung heraus über das Menü ?|Microsoft Visual Basic-Hilfe oder die F1-Taste zu erreichen. Innerhalb von Modulblättern liefert das Drücken von F1 eine Kontexthilfe zum vom Cursor markierten Stichwort. Eine weitere bequeme und schnelle Möglichkeit die Visual Basic-Hilfe zu einem bestimmten Stich- oder Schlüsselwort aufzurufen, erhält man durch klicken der Fragenzeichen-Schaltfläche des Objektkataloges.

2. Makroaufzeichnung – Excel macht's selbst

Arbeitet man intensiv mit Excel, so stellt man fest, dass immer wieder dieselben oder ähnliche Abfolgen von Befehlen auftreten, z. B. das Erstellen immer gleicher Listen oder das Erzeugen bestimmter sich nicht verändernder Tabellenköpfe.

Derartige Aktionen können mit Hilfe von Makros automatisiert und so die Arbeit mit Excel wesentlich erleichtert und beschleunigt werden.

Doch was haben Makros in einer Einführung von Visual Basic for Applications zu suchen? Ganz einfach. Makros bezeichnen hier eine Reihe von Anweisungen an Excel, das diese ausführt, sobald es dazu aufgefordert wird. Bis Excel 4.0 gab es eine eigene Makroprogrammiersprache, die aus Kompatiblitätsgründen noch in allen späteren Versionen von Excel verstanden wird (sog. Excel-4-Makros). Mit Excel 5.0 wurde erstmals Visual Basic for Applications eingeführt und dient seitdem als Makroprogrammiersprache (sog. VBA-Makros).

Grundsätzlich bestehen zwei Möglichkeiten, Makros zu erzeugen: Entweder werden sie vom Benutzer eingetippt oder man veranlaßt Excel ein Makro aufzuzeichnen. Dabei werden die Arbeitsschritte des Benutzers verfolgt und als VBA-Anweisungen in einem Modul zur Arbeitsmappe bereitgestellt. Beim Ausführen des Makros wiederholt Excel exakt die gleichen Aktionen des Benutzers.

2.1. Makros aufzeichnen und ausführen

Die Aufzeichnung von Makros durch Excel erreicht man durch den Menübefehl Extras|Makros|Aufzeichnen. . . , wodurch man aufgefordert wird den eindeutigen Namen und eine Tastenkombination zur Ausführung des Makros festzulegen. In einer Auswahlliste bestimmen Sie die Gültigkeit des Makros. Wählt man 'Makro speichern in: Persönliche Makroarbeitsmappe', so ist das Makro global in allen Arbeitsmappen verfügbar. Wird dagegen 'Makro speichern in: Diese Arbeitsmappe', so ist das Makro nur innerhalb der aktiven Arbeitsmappe gültig. Im Anschluss daran werden alle Arbeitsschritte durch Excel aufgenommen, solange bis die Makroaufnahme beendet wird.

Nach der Aufnahme eines Makros wird es in der Liste aller verfügbaren Makros dieser Arbeitsmappe aufgelistet und kann von hier aus ausgeführt, bearbeitet und gelöscht werden. Die Übersicht über die verfügbaren Makros erhält man über Extras|Makro|Makro. . . . Darüber hinaus kann ein Makro auch durch die ihm zugewiesene Tastenkombination, Symbolschalftfläche oder einen Menüpunkt ausgeführt werden, was wesentlich bequemer ist.

2.2. Makros bearbeiten

Nachdem ein Makro aufgenommen wurde, steht es innerhalb der Entwicklungsumgebung in einem von Excel eigens erzeugten Modul bereit. Mit Hilfe von Visual Basic kann dieses Makro nun bearbeitet, angepasst und optimiert werden. Aus der Entwicklungsumgebung heraus können Makros im übrigen immer mit F5 ausgeführt werden. Dabei wird dasjenige Makro ausgeführt, das gerade den Cursor besitzt.

3. Visual Basic for Applications – Sprachgrundlagen

In diesem Abschnitt werden grundlegende Elemente der Programmiersprache Visual Basic erläutert. Dazu gehören Erklärungen zu Datentypen, Variablen, Prozeduren, Schleifen und Verzweigungen durch Bedingungen. Auf die objektorientierte Programmierung wird hier noch nicht eingegangen, da lediglich Sprachgrundwissen bereitgestellt werden soll.

3.1. Variablendeklaration und Datentypen

3.1.1. Variablen und Konstanten

Analog zu anderen Programmiersprachen werden Variablen verwendet, um zur Programmlaufzeit bestimmte Daten zur späteren Verwendung zwischenzuspeichern. Die Variablendefinition erfolgt in VBA mit dem Schlüsselwort ***Dim*** gefolgt von einem Variablennamen. Für die Wahl des Variablennamens sind allerdings bestimmte Einschränkungen zu berücksichtigen. So darf der Name nicht länger als 255 Zeichen sein, nicht mit einem in Visual Basic vordefiniertem Schlüsselwort übereinstimmen und keine Leerzeichen, Punkte oder Sonderzeichen enthalten. VBA achtet außerdem nicht auf Groß- und Kleinschreibung. Allerdings dürfen Variablennamen hier deutsche Sonderzeichen (ä, ü, ö, ß) enthalten.

Im Anschluss an den Variablennamen sollte immer der Datentyp der Variablen angegeben werden. Die Typzuweisung erfolgt über das Schlüsselwort ***As*** gefolgt von einem Typbezeichner. Eine Variablendeklaration sollte also immer folgendes Aussehen haben: `Dim` *Variablenname* `As` *Variablentyp*.

Im Gegensatz zu Variablen ändern Konstanten über die gesamte Laufzeit des Programms nicht ihren Wert. Sie werden über das Schlüsselwort ***Const*** definiert gefolgt von einem Konstantennamen und einer Wertzuweisung. Für den Namen gelten die gleichen Restriktionen wie für Variablennamen. Zudem hat man die Möglichkeit typisierte Konstanten festzulegen, d. h. neben der Wertzuweisung wird für die Konstanten auch ein Datentyp festgelegt. Die Definition einer Konstanten sollte daher wie folgt aussehen: `Const` *Konstantenname* `=` *Wert* bzw. `Const` *Konstantenname* `As` *Konstantentyp* `=` *Wert*.

3.1.2. Verfügbare Datentypen

Datentypen definieren für eine Klasse von Daten einen Wertebereich, den zur Speicherung des Datenwertes benötigten Speicherplatz und legen darüber hinaus die Operationen fest, die auf diese Daten angewendet werden können.

In Tabelle 1 sind die in Visual Basic verfügbaren Standard-Datentypen aufgelistet. Neben der Variablendeklaration ***Dim...As...*** ist ebenfalls eine Deklaration mit Hilfe der in der letzten Spalte angegebenen Kurzschreibweisen möglich. Dazu wird das jeweilige Kennungszeichen dem Variablennamen direkt angefügt, z. B. zur Definition einer Integerzahl `Dim Zahl%`.

3.1.3. Definition eigener Datentypen

Wie in anderen Programmiersprachen ist es auch in Visual Basic möglich, die vorhandenen Datentypen zu neuen Datenstrukturen zusammenzusetzen. Ein solches Konstrukt wird benutzerdefinierter Datentyp (engl. UDT = user defined data type) genannt. Die Definition eines solchen Datentyps wird durch das Schlüsselwort ***Type*** eingeleitet und durch ***End***

Ganze Zahlen	**Byte**	Wertebereich: 0...255, 1 Byte Speicherbedarf	
	Integer	Wertebereich: -32 768...32 767, 2 Byte Speicherbedarf	%
	Long	Wertebereich: -2 147 483 648...2 147 483 647, 4 Byte Speicherbedarf	&
Reelle Zahlen	**Single**	Fließkommazahl mit 8 Stellen Genauigkeit, 4 Byte Speicherbedarf	!
	Double	Fließkommazahl mit 16 Stellen Genauigkeit, 8 Byte Speicherbedarf	#
	Currency	Festkommazahlen mit 15 Stellen Genauigkeit vor und 4 Stellen hinter dem Komma; Speicherbedarf: 8 Byte	
Zeichenketten	**String**	Speicherbedarf: 10 Byte und 2 Byte pro Zeichen; Länge nur durch RAM begrenzt	$
Wahrheitswerte	**Boolean**	Datentyp, der nur zwei Werte annehmen kann (True, False); Speicherbedarf: 2 Byte	
Variant-Datentyp	**Variant**	nimmt je nach Bedarf einen der obigen Datentypen an; Speicherbedarf: mindestens 16 Byte, bei Zeichenketten 22 Byte plus 2 Byte pro Zeichen	

Tabelle 1: Standard-Datentypen in Visual Basic

Type abgeschlossen. Innerhalb dieses Blocks können beliebig viele Variablendeklarationen aufgeführt werden, jeweils in der Form *variablenname As variablentyp*. Der Zugriff auf einzelne Variablen der Struktur erfolgt durch Nachstellen eines Punktes gefolgt vom gewünschten Elementnamen. Das folgende Beispiel veranschaulicht die Erzeugung und Verwendung eines benutzerdefinierten Datentyps:

```
Type TBaum
  Name As String
  MaxAlter As Integer
  IstNadelbaum As Boolean
End Type
Sub Testmakro()
  Dim Baum1 As TBaum
  Dim Baum2 As TBaum
  Baum1.Name = "Ahorn"
  Baum1.IstNadelbaum = False
  Baum1.MaxAlter = 90
  Baum2 = Baum1
  Baum2.Name = "Birke"
End Sub
```

3.1.4. Gültigkeitsbereiche von Variablen

Generell unterscheidet man zwischen globalen und lokalen Variablendeklarationen. Lokale Variablen werden immer innerhalb von Prozeduren mit Hilfe des Schlüsselworts Dim definiert und können auch nur dort verwendet werden. Außerhalb der Prozedur ist die Variable nicht bekannt.

Globale Variablen sind immer global mindestens innerhalb des Modulblattes gültig. Solche Variablen heißen daher Modulvariablen und werden im Modulkopf mit den Schlüsselwörtern ***Dim*** bzw. ***Private*** definiert. Wird anstelle von *Dim* oder *Private* das Schlüsselwort ***Public*** verwendet, so definiert man sog. öffentliche Variablen, die in allen Modulblättern der aktuellen Arbeitsmappe Gültigkeit besitzen.

Die Inhalte globaler Variablen sind solange verfügbar, bis die den Modulblättern übergeordnete Arbeitsmappe geschlossen wird. Lokale Variablen sind dagegen nur solange mit gültigem Inhalt gefüllt, bis die Ausführung der Prozedur beendet ist.

3.1.5. Option Explicit – Eine Empfehlung

Die Verwendung von Variablen in Visual Basic lässt dem Benutzer (leider) sehr viel Freiheiten. Es ist erlaubt, Variablen ohne vorherige Deklaration und Typzuweisung zu benutzen. In der Regel schleichen sich bei dieser Vorgehensweise aber schnell Fehler ein, die dann nur schwer zu finden sind. Daher ist es grundsätzlich zu empfehlen, Variablen vor der Benutzung durch den ***Dim***-Befehl zu deklarieren.

Mit Hilfe der Anweisung *Option Explicit* am Beginn des Moduls weist man Visual Basic an, auf die Variablendeklaration zu bestehen. Excel weigert sich dann nämlich, eine Prozedur auszuführen, solange es nicht alle darin enthaltenen Variablennamen kennt.

Wird in Extras|Optionenn|Modul allgemein die Option „Variablendeklaration erforderlich“ aktiviert, so fügt Excel automatisch in jedes neue Modulblatt die Anweisung *Option Explicit* ein.

3.2. Felder

Felder sind Listen von Variablen gleichen Datentyps. Sie werden zur Verwaltung von Daten ähnlichen Inhalts verwendet. Auf die einzelnen Elemente eines Feldes wird mittels Indizes zugegriffen. In Visual Basic unterscheidet man dimensionierte und dynamische Felder.

3.2.1. Dimensionierte Felder

Dimensionierte oder auch statische Felder sind Datenreihen fester Länger, d. h. mit einer vorher festgelegten Anzahl von Elementen. Die Definition eines dimensionierten Feldes erfolgt mit dem Schlüsselwort *Dim*, dem Feldnamen und der Länge des Feldes in Klammern als größten verfügbaren Index. Über *As* wird im Anschluss daran ein Datentyp angegeben, den alle Elemente des Feldes annehmen. Felder sind standardmäßig nullbasiert, d. h. das erste Element des Feldes wird über den Index 0 angesprochen. Mehrdimensionale Felder erhält man, indem innerhalb des Klammerpaares alle Dimensionen (d. h. jeweils der größte Index) durch Kommata getrennt aufgeführt werden. Für den Zugriff auf ein Element wird nach dem Feldnamen in Klammern der Index des gewünschten Elements angegeben. Die Erzeugung und Verwendung statischer Felder soll im folgenden Beispiel gezeigt werden:

```
Dim meinfeld(10) As Integer          'Integer-Feld mit 11 Elementen
Dim kurzfeld$(3)                     'String-Feld in Kurzschreibweise
Dim feld(-5 To 4)                    'Indexbereich selber vorgeben
Dim matrix(4,4) As Single            'zweidimensionales Feld (5x5)
meinfeld(0) = 0: meinfeld(1) = 0
kurzfeld(0) = "Erster Eintrag"
```

```
kurzfeld(3) = "Letzter Eintrag"
matrix(0,0) = 1: matrix(0,1) = 0
```

3.2.2. Dynamische Felder

Im Gegensatz zu statischen Felder wird für dynamische Felder zum Definitionszeitpunkt keine Länge vorgegeben. Die Deklaration als dynamisches Feld erfolgt analog zu dimensionierten Felder, jedoch hier durch ein leeres Klammerpaar. Zur Laufzeit wird mit Hilfe des ***ReDim***-Befehls die benötigte Feldlänge eingestellt, die jederzeit wieder verändert werden kann. Mit Hilfe des Schlüsselwortes ***Preserve*** erreicht man zusätzlich, dass bei der Umdimensionierung des Feldes der bisherige Inhalt erhalten wird. Dynamische Felder können beliebig groß und in beliebig vielen Dimensionen definiert werden. Die Verwendung dynamischer Felder verdeutlicht das nächste Beispiel:

```
Dim dynfeld() As Integer            'dynamisches Integer-Feld
ReDim dynfeld(anzahl)               'Feldgröße anpassen
ReDim Preserve dynfeld(anzahl + 5)  'Feldgröße erneut anpassen
```

3.2.3. Umgang mit Feldern

Mit Hilfe der Funktionen ***LBound*** und ***UBound*** können für Felder die untere bzw. obere Indexgrenze ermittelt werden. Dieser Zugriff auf die Indexgrenzen eignet sich ganz besonders zur Verwendung von Schleifen im Zusammenhang mit Feldern.

Bei statischen Feldern können mit der Anweisung ***Erase*** in einem Befehl die Inhalte aller Elemente eines Feldes gelöscht werden. Wird der *Erase*-Befehl auf dynamische Felder angewendet, so wird das gesamte Feld gelöscht und der vom Feld belegte Speicherplatz freigegeben. Vor der weiteren Verwendung muss das Feld dann mit dem *ReDim*-Befehl neu dimensioniert werden.

3.3. Prozeduren – Unterprogramme und Funktionen

Das Zentrale der Prozeduralen Programmierung liegt in der Unterteilung des Quellcodes in kleine voneinander getrennte „Codehäppchen“ – sog. Prozeduren. Diese Programmteile können sich gegenseitig aufrufen und einander Daten in Form von Parametern übergeben. Man unterscheidet in Visual Basic genau wie in anderen gängigen prozeduralen Programmiersprachen zwischen Prozeduren ohne Rückgabewerts (*subs*) und Prozeduren mit Rückgabewert (*functions*).

3.3.1. Sub-Definition und -Aufruf

Sub-Prozeduren definieren durch eine Abfolge von Anweisungen ein Unterprogramm und liefern als entscheidendes Merkmal keinen Rückgabewert. Sie werden durch ***Sub*** *name(Parameterliste)*... ***End Sub*** erzeugt. Werden keine Parameter übergeben, so muss ein leeres Klammerpaar gesetzt werden. Der Aufruf einer solchen Prozedur erfolgt durch den Prozedurenamen gefolgt von einer Liste mit Parametern. Die Parameterliste wird dabei nicht in Klammern angegeben, sondern durch ein Leerzeichen vom Namen getrennt angefügt.

```
Sub MeinMakro(Parameter1, Parameter2)   'Sub-Definition
```

```
    ...                                     'Folge von Anweisungen
End Sub

Sub TesteMeinMakro()
    MeinMakro Parameter1, Parameter2    'Aufruf von MeinMakro
End Sub
```

Das vorzeitige Verlassen eines Unterprogrammes erfolgt durch die Anweisung ***Exit Sub***.

3.3.2. Funktionsdefinition und -aufruf

Funktionen definieren ebenfalls einen Block von Anweisungen mit der Besonderheit, dass ein Ergebnis zurückgeliefert wird. Der Rückgabewert muss vor dem Verlassen der Funktion dem Funktionsnamen mitgeteilt werden. Funktionen werden durch ***Function*** *name(Parameterliste)* ***As*** *typ...* ***End Function*** erzeugt. Dem Funktionsnamen und der Parameterliste wird hier noch der Datentyp des Rückgabewertes angegeben.

Der Aufruf einer Funktion erfolgt anders als der Aufruf eines Unterprogramms, da hier ein Rückgabewert geliefert wird, der einer Variablen zugewiesen werden muss. Außerdem werden beim Aufruf der Funktion die Parameter dem Funktionsnamen in Klammern angehängt.

```
Function MeineFunktion(Parameter1) As Long
    ...
    MeineFunktion = Ergebnis
End Sub

Sub TesteMeineFunktion()
    Dim Wert As Long
    Dim Par1
    ...
    Wert = MeineFunktion(Par1)
End Sub
```

Statt der Typzuweisung über *As* kann hier analog zur Variablendeklaration die Kurzschreibweise verwendet werden, im Beispiel `Function MeineFunktion&(Parameter1)`.

Soll die Funktion vor ihrer vollständigen Abarbeitung verlassen werden, verwendet man die Anweisung ***Exit Function***.

3.4. Parameter

Um Prozeduren Daten zur weiteren Verwendung zu übergeben, muss die Parameterliste gesetzt werden. Dies geschieht durch das Setzen von Platzhaltern, denen zum Zeitpunkt des Aufrufs der Prozedur Variablenwerte oder Verweise auf Variablen zugewiesen werden. Grundsätzlich sollte aus Gründen der Fehlervermeidung und Effizienz auch zur Definition der Parameter immer der jeweilige Datentyp angegeben werden.

```
Function TestFunk(Par1 As Long, Par2 As Integer) As Long
Funktion TestFunk&(Par1&, Par2%)
```

Man unterscheidet grundsätzlich zwischen *Werteparametern* und *Referenzparametern*.

3.4.1. Werteparameter

Werteparameter werden durch das Schlüsselwort ***ByVal*** innerhalb der Parameterliste erzeugt. Die Besonderheit liegt hierbei darin, dass der Prozedur lediglich der Wert der übergebenen Variablen übermittelt wird. Das hat zur Folge, dass die Variable durch den Code der Prozedur nicht verändert werden kann; eine Veränderung des Wertes innerhalb der Prozedur wirkt sich nicht auf den Wert der Variablen außerhalb der Prozedur aus.

```
Function GetCelsius(ByVal fGrad As Double) As Double
```

3.4.2. Referenzparameter

Referenzparameter erzeugt man durch das Schlüsselwort ***ByRef***. Referenzparameter sind die Standardeinstellung in Visual Basic. Das Schlüsselwort *ByRef* kann daher fallen gelassen werden. Der Prozedur wird im Falle eines Referenzparameters nicht der Variablenwert übergeben, sondern ein Verweis (die Referenz) auf die Variable, d. h. die Adresse der Variablen im Speicher. Dies hat zur Folge, dass im Prozedurinneren ein Ändern des Variablenwertes auch die Veränderung des Wertes außerhalb der Prozedur nach sich zieht.

```
Sub KillSpace(ByRef Text As String)
```

3.4.3. Felder als Parameter

Neben den gängigen Datentypen können auch Felder als Parameter in Prozeduren auftreten. Dies erreicht man, indem der Parameter in der Parameterliste durch ein angehängtes Klammerpaar als Feld gekennzeichnet wird. Zu beachten ist, dass Feldparameter immer Referenzparameter sind. Das Schlüsselwort *ByVal* ist hier daher nicht erlaubt. Weiterhin muss der Variablentyp des zu übergebenen Feldes immer mit dem Typ des Feldparameters übereinstimmen. Im folgenden Beispiel wird der Prozedur *ClearFeld* ein Integer-Feld als Parameter übergeben, die alle Elemente des Feldes auf Null setzt.

```
Sub ClearFeld(feld() As Integer)
    Dim i As Integer
    For i = LBound(feld) to UBound(feld)
        feld(i) = 0
    Next i
End Sub
...
Sub TestMakro()
    Dim MeinFeld(10) As Integer
    ...
    ClearFeld(MeinFeld)
    ...
End Sub
```

3.5. Gültigkeitsbereich von Prozeduren

Im Unterschied zu Variablen und Konstanten, die ohne zusätzliche Angaben lediglich im jeweiligen Modulblatt definiert sind, sind Prozeduren generell öffentlich, man kann sie al-

so innerhalb der gesamten Arbeitsmappe verwenden. Existieren in verschiedenen Modulen gleichnamige Prozeduren, so muss der Modulname mit Punkt vorangestellt werden.

Soll verhindert werden, dass eine Prozedur außerhalb des Moduls, in dem sie definiert wurde, verwendet wird, muss das Schlüsselwort ***Private*** der Prozedurdefinition vorangestellt werden. Die Verwendung des *Private*-Schlüsselworts hat außerdem zur Folge, dass der Prozedurname nicht in der Makroliste der Arbeitsumgebung (Extras|Makros) auftaucht, was dort zu einer größeren Übersichtlichkeit führt.

3.6. Verzweigungen/Bedinungen

Verzweigungen und Bedingungen dienen der Strukturierung von Programmcode und erlauben das wahlweise Ausführen bestimmter Programmteile. Visual Basic stellt zwei verschiedene Arten für Verzweigungen zur Verfügung: die ***If-Then-Else***-Anweisung und die ***Select-Case***-Anweisung.

3.6.1. If-Then-Else

Die *If*-Abfrage beginnt grundsätzlich mit ***If*** *bedingung* ***Then*** gefolgt von einem Anweisungsblock und endet schließlich mit ***End If***. Der Anweisungblock wird nur dann ausgeführt, wenn die Bedingung erfüllt ist, andernfalls wird sie übergangen, falls durch *Else If bedingung* oder ***Else*** kein weiterer Anweisungsteil angegeben wurde. Die Syntax sieht also wie folgt aus:

```
If Bedingung1 then
    Anweisung1
Else If Bedingung2 then
    Answeisung2
Else
    Standardbehandlung
End If
```

3.6.2. Select-Case

Neben der If-Abfrage existiert eine weitere, in vielen Fällen übersichtlichere Verzweigungsstruktur – die ***Select-Case***-Abfrage. Bei dieser wird ein gegebener Ausdruck untersucht und in Abhängigkeit von den möglichen Ausprägungen ein bestimmter Anweisungsblock ausgeführt.

```
Select Case Ausdruck
Case Ausprägung1
    Anweisung1
Case Ausprägung2
    Anweisung2
Case Else
    Standardbehandlung
End Select
```

3.7. Schleifen

Schleifen ermöglichen es, Programmteile mehrfach hintereinander zu wiederholen. Visual Basic stellt zwei grundlegende Schleifentypen zur Verfügung – die ***For-Next***-Schleife und die ***Do-Loop***-Schleife mit ihren jeweiligen Abwandlungen *For-Each-Next* bzw. *While-Wend*.

3.7.1. For-Next

Die *For-Next*-Schleife stellt die am einfachsten zu verwendende Schleife dar. Einer Zählvariablen wird dabei ein Anfangswert zugewiesen und von Durchgang zu Durchgang hochgezählt. Die Schleife wird solange wiederholt bis die Zählvariable den angegebene Endwert überschreitet.

```
Dim i As Integer
For i = 1 To 100
    ...
    If Bedingung Then Exit For
Next i
```

Entfällt die ***Step***-Angabe im Anschluss an den Endwert, so wird standardmäßig die Schrittweite 1 beim Hochzählen der Laufvariablen angenommen. Wie im Beispiel zu sehen ist, kann mit der Anweisung ***Exit For*** die Schleife jederzeit verlassen werden.

3.7.2. For Each-Next

Die ***For Each-Next***-Schleife stellt eine Sonderform der *For-Next*-Schleife da. Sie eignet sich besonders im Umgang mit Feldern und Aufzählmethoden. Einer Variablen wird in jedem Schleifendurchlauf ein Element eines Feldes oder einer Aufzählungmethode übergeben.

```
Dim Element
Dim Feld() As Integer
For Each Element In Feld()
    Debug.Print Element
Next Element
```

Zu beachten ist, dass der Schleifenvariablen (hier *Element*) der Variant-Typ zugewiesen werden muss.

3.7.3. Do-Loop

Die *Do-Loop*-Schleife ist eine sehr flexible Schleifenart, die ausgeführt wird, solange oder bis eine bestimmte Bedingung erfüllt ist. Sie wird durch das Schlüsselwort ***Do*** eingeleitet und mit ***Loop*** beendet. Mit den Schlüsselwörtern ***While*** bzw. ***Until*** wird die Bedingung angegeben, die bestimmt solange oder bis wann die Schleife wiederholt wird. *While/Until* dürfen dabei entweder dem *Do*-Schlüsselwort oder dem *Loop*-Schlüsselwort folgen. Im ersten Fall wird zuerst immer die Bedingung geprüft und anschließend die Schleife ausgeführt, im zweiten Fall wird die Schleife auf jeden Fall durchlaufen und im Anschluss daran erst die Abbruchbedingung geprüft.

```
Do While/Until Bedingung
    Anweisungen
Loop

'oder so

Do
    Anweisungen
Loop While/Until Bedingung
```

Analog zur *For-Next-* bzw. *For Each-Next*-Schleife kann auch die *Do-Loop*-Schleife mittels ***Exit Do*** verlassen werden.

3.7.4. While-Wend

While-Wend-Schleifen sind identisch zu *Do While-Loop*-Schleifen mit dem einzigen Unterschied, dass sie nicht durch *Exit* verlassen können. Die Syntax sieht folgendermaßen aus:

```
Dim Länge As Byte
While Länge <= 20
    BerechneNeu(Länge)
Wend
```

4. Objektorientierte Programmierung – Eine Einführung

4.1. Was ist ein Objekt

Bisher haben wir uns mit der prozeduralen Programmierung beschäftigt, mit strikter Trennung zwischen Daten und Prozeduren. In der objektorientierten Programmierung (OOP) werden Daten und Prozeduren zusammengefasst zu inhaltlich abgeschlossenen Strukturen – zu *Objekten*. Das bedeutet auch, dass Daten eines Objektes nur durch objekteigene Prozeduren geändert werden können (OOP-Stichwort: *Kapselung*).

In der Realität besitzen Objekte bestimmte Attribute (Eigenschaften) mit entsprechenden Werten, außerdem bestimmte Verhaltensweisen. Ein Beispiel soll das veranschaulichen:

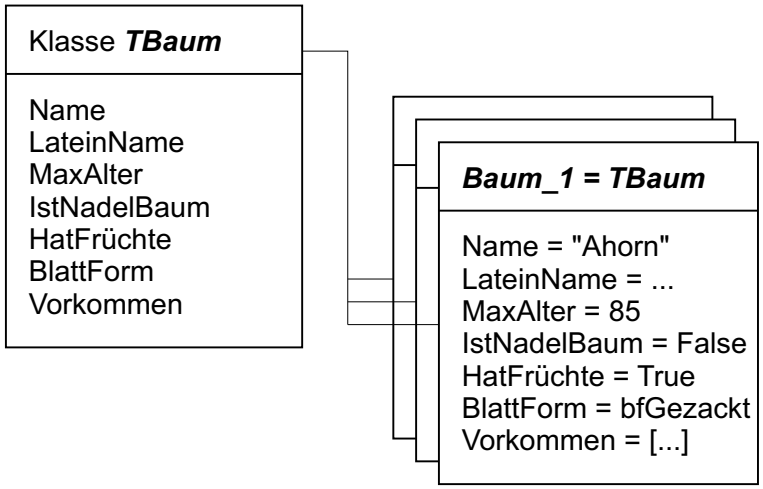

Abbildung 6: Objektklasse mit Objekten

In der objektorientierten Programmierung werden solche realen Objekte abstrakt dargestellt. Man bezeichnet die Attribute als *Objekteigenschaften* und die zugehörigen Verhaltensweisen als *Objektmethoden*. Darüber hinaus definieren Objekte bestimmte *Ereignisse*, auf die reagiert werden kann.

Objekte werden in *Klassen* zusammengefasst. In einer Klasse wird die Struktur eines Objekts festgelegt, d. h. die Klasse beschreibt, welche Eigenschaften und Verhaltensweisen (Methoden) einem Objekt dieser Klasse zugrundeliegen.

4.2. Objekte in Excel

In VBA werden als Objekte Elemente von Excel bezeichnet. Dazu gehören beispielsweise *Application* (Anwendungshauptfenster, also Excel selber), *Workbook* (Arbeitsmappe), *Window* (Fenster), *Worksheet* (Arbeitsblatt innerhalb einer Arbeitsmappe), *Range* (Bereichsobjekt zum Zugriff auf Zellen eines Tabellenblattes) u. v. m. In Excel sind insgesamt etwa 150 Objekte definiert, die in einer hierarchischen Struktur vorliegen. An oberster Stelle befindet sich das Application-Objekt, das alle weiteren Objekte unter sich vereint.

Die Arbeit mit Objekten in Excel wird erst dann sinnvoll, wenn spezifische Daten der Objekte gelesen oder geändert werden, neue Objekte angelegt und gelöscht (wieder freigegeben) werden können. Dazu stellt jedes Objekt verschiedene Eigenschaften und Methoden zur Verfügung.

Um mit einem Objekt einer Klasse arbeiten zu können, d. h. Eigenschaftswerte des Objekts festlegen und Objektmethoden aufrufen zu können, muss eine neue *Instanz* (Exemplar) für dieses Objekt angelegt werden. Dabei wird im Speicher des Computers Platz für die gesamte Struktur des Objekts reserviert, so wie sie von der Klassendefinition vorgegeben wird. Über eine *Objektvariable* kann nun auf die Instanz des Objekts zugegriffen werden und Daten des Objekts gesetzt bzw. geändert werden. Um in VBA Instanzen selbstdefinierter Objekte anzulegen, wird das Schlüsselwort ***New*** verwendet.

Sollen Objektvariablen Werte (in diesem Fall Objekte) zugewiesen werden, so muss im Unterschied zur Wertzuweisung bei gewöhnlichen Variablen das Schlüsselwort ***Set*** verwendet werden. Weiterhin handelt es sich bei Objektvariablen immer um Verweise (Zeiger) auf den Speicherbereich, in dem das Objekt instanziiert (angelegt) wurde. Ein Verweis auf ein Objekt kann durch das Schlüsselwort ***Nothing*** gelöscht werden.

```
Dim MyObject As TMyObject       'Objektvariable festlegen
Set MyObject = New TMyObject    'Eine Instanz erzeugen
MyObject.Methode1 param1,param2
MyObject.Eigenschaft
...
Set MyObject = Nothing          'Objektvariable freigeben
```

Über die Funktion ***TypeName*** kann der Typ einer Objektariablen ermittelt werden. *TypeName* liefert dabei als Ergebnis eine Zeichenkette zurück, z. B. `"Workbook"` oder `"Window"`.

4.2.1. Objekteigenschaften und -methoden

Eigenschaften bestimmen Merkmale eines Objekts, z. B. die Schriftart innerhalb einer Tabellenzelle. Objekteigenschaften sind im Wesentlichen nichts anderes als Variablen, in diesem Fall spezielle Variablen, die genau einem Objekt angehören. Aus diesem Grund muss beim Zugriff auf die Eigenschaft immer das Objekt angegeben werden.

```
ActiveCell.Font.Name = "Arial"
```

Im Beispiel wird mit Hilfe der Eigenschaft *Font* des *ActiveCell*-Objekts die Schriftart geändert. Die Eigenschaft *Font* ist hierbei ein eigenes Unterobjekt von *ActiveCell* mit einer Eigenschaft *Name*, die den Namen der gewählten Schriftart angibt.

Methoden sind objekteigene Prozeduren. Methoden definieren also eine Reihe von Anweisungen, die für das Objekt ausgeführt werden sollen. Man unterscheidet genau wie bei gewöhnlichen Prozeduren zwei Typen: Methoden, die keinen Rückgabewert liefern und Methoden die ein Ergebnis zurückgeben.

```
Cells("A1").Select                      'Methode ohne Rückgabewert
Set MyWorksheet = Worksheets.Add        'Erzeugen einer neuen Tabelle
MyWorksheet.Name = "Mein Tabellenblatt" 'Eigenschaft Name setzen
```

Eine wichtige Menge von Methoden in Excel sind die sog. Aufzählmethoden (*Workbooks*, *Worksheets*, *Cells*). Mit ihnen kann auf eine Gruppe gleichartiger Unterobjekte zugegriffen werden und sie eignen sich besonders zur Anwendung in *For Each*-Schleifen. Verwendet man die Aufzählmethoden mit Parameter, so wird ein bestimmtes Objekt der Aufzählung zurückgegeben. Im folgenden Beispiel wird auf das fünfte Element der Aufzählung Worksheets

zugegriffen, also das fünfte Tabellenblatt zurückgegeben. Zu beachten ist, dass die Indizierung bei Aufzählungen im Gegensatz zu Feldern standardmäßig mit 1 beginnt. Der Index des ersten Elements lautet immer 1 (nicht 0).

```
Dim MyWorksheet As Worksheet        'Objektvariable anlegen
Set MyWorksheet = Worksheets(5)     'fünftes Aufzählobjekt zuweisen
```

Werden Aufzählmethoden ohne Parameter verwendet, werden sie wie ein Objekt behandelt, verweisen also auf die Sammlung der Aufzählungselemente. Für diese Aufzählobjekte existieren unabhängig von der Objektklasse der Elemente bestimmte Eigenschaften und Methoden: die Anzahl der Elemente wird beispielsweise mit ***Count*** bestimmt, mit ***Add*** werden neue Elemente der Aufzählung hinzugefügt und mit ***Delete*** entfernt.

Mit Hilfe des Schlüsselwortpaares ***With*** . . . ***End With*** hat man die Möglichkeit schneller und übersichtlicher auf bestimmte Objekteigenschaften und -methoden zuzugreifen, anstatt jeweils alle Objektangaben vor den jeweiligen Bezeichner zu setzen. Mit Hilfe von *With* kann eine Menge Schreibarbeit gespart werden, indem mehrere Eigenschaften und Methoden eines Objektes innerhalb eines Blocks angesprochen werden. Das folgende Beispiel gibt eine Gegenüberstellung:

```
Selection.Font.Name = "Arial"
Selection.Font.Size = 10
Selection.Font.Bold = True
Selection.Font.Italic = False
Selection.Font.Underline = False

With Selection.Font
    .Name = "Arial"
    .Size = 10
    .Bold = True
    .Italic = False
    .Underline = False
End With
```

4.2.2. Objektereignisse

Im Rahmen dieses Kapitels wurde bisher nur auf die Objektorientierung von VBA eingegangen. VBA ist darüberhinaus aber auch eine ereignisorientierte Programmiersprache. Ereignisorientiert meint hierbei, dass Programmcode als Reaktion auf bestimmte Ereignisse ausgeführt werden kann. Ereignisse werden dabei zumeist durch den Anwender hervorgerufen (z. B. Mausklicks, Tastatureingaben).

Neben Eigenschaften und Methoden werden also innerhalb einer Klasse zusätzlich Ereignisse festgelegt. Der Benutzer hat die Möglichkeit durch das Schreiben sog. Ereignisbehandlungsprozeduren auf das Eintreten dieser Ereignisse zu reagieren.

In Excel 5/7 unterscheidet sich das Bereitstellen von Code zur Reaktion auf bestimmte Ereignisse leider von der Art und Weise wie ab Excel 97 auf Ereignisse reagiert wurde.

Zu jedem Objekt gab es in Excel 5/7 Ereignisse in der Form *OnEventXxx*. Um auf ein solches Ereignis reagieren zu können, musste man diesen Ereignissen eine Prozedure zuweisen. Ein Beispiel soll das verdeutlichen:

```
Sub Reagiere()
    MsgBox "Ein Ereignis ist eingetreten..."
End Sub
...
Worksheets(1).OnSheetActivate = "Reagiere"
```

Die Ereignisbehandlung ab Excel 97 wurde völlig überarbeitet. Jede Ereignisbehandlungsprozedure besitzt nun einen eindeutigen Namen, der sich aus dem Objektnamen, einem Unterstrich und dem Ereignisnamen zusammensetzt – etwa `Worksheet_Activate`. Existiert eine solche Prozedure, so wird sie von Excel automatisch ausgeführt.

Die Erzeugung solcher Prozeduren zur Reaktion auf Ereignisse speziell zur Arbeitsmappe und deren Arbeitsblättern ist sehr einfach geworden, da innerhalb der Entwicklungsumgebung dafür entsprechende Modulblätter vorgesehen sind (Doppelklick im Objektfenster auf die Untereinträge zur aktuellen Arbeitsmappe). In der Kopfleiste dieser Modulblätter können zum entsprechenden Objekt (Arbeitsmappe, Arbeitsblatt) die zugehörigen Ereignisse komfortabel über eine Auswahlliste markiert und der Behandlungscode bereitgestellt werden.

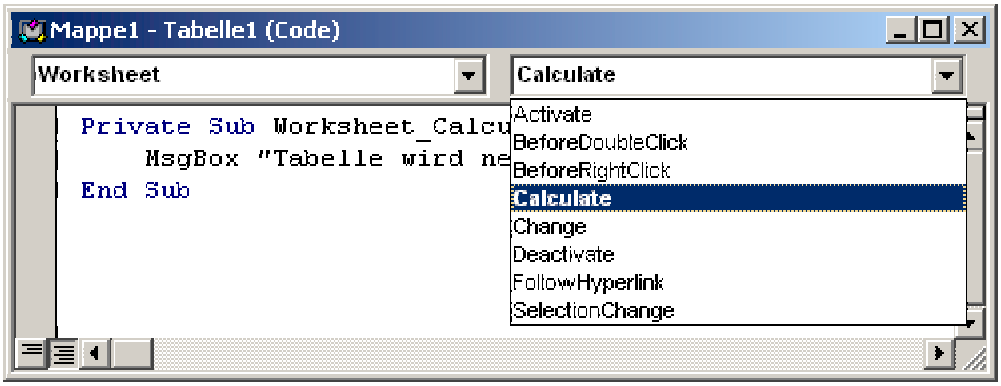

Abbildung 7: Auswahl der Ereignisprozeduren im Codefenster

Das Reagieren auf Ereignisse anderer Objekte, z. B. Application gestaltet sich ungleich schwieriger. Verwendung findet hierbei das Schlüsselwort *WithEvents*. Für nähere Informationen sei auf die Online-Hilfe von Excel verwiesen.

4.2.3. Eigene Objektklassen erzeugen

Excel gestattet dem Benutzer eigene Objektklassen zu definieren und zu verwenden. Dazu muss einer Arbeitsmappe im Objektfenster der Entwicklungsumgebung ein sog. Klassenmodul hinzugefügt werden, was den Programmcode zur Definition der neuen Klasse aufnimmt. Im Eigenschaftsfenster wird die Eigenschaft *Name* der neuen Objektklasse festgelegt und im Anschluss daran kann die Klasse mit Objekteigenschaften und -methoden ausgestattet werden[1].

[1] siehe Kofler, Michael *VBA-Programmierung mit Excel 97*, Addison-Wesley-Longman, Bonn, 1997, Kap. 5.3.5 Klassenmodule, S. 142ff

5. Wichtige Objekte in Excel

Im Folgenden soll in einer kurzen, übersichtlichen Weise auf einige für die VBA-Programmierung wichtige Objekte in Excel eingegangen werden.

5.1. Bereichsobjekte

Bereichsobjekte ermöglichen in Excel den Zugriff auf Zellbereiche/Zellen eines Arbeitsblattes. Leider ist der Umgang mit Bereichsobjekten zunächst verwirrend, da gleiche Begriffe für inhaltlich unterschiedliche Dinge verwendet werden.

5.1.1. Das Range-Objekt

Zentrale Bedeutung beim Zugriff auf Daten der Arbeitsmappen hat das Range-Objekt. Es beschreibt Bereiche von Zellen oder einzelne Zellen. VBA kennt in diesem Zusammenhang kein eigenes Zell-Objekt zum Zugriff auf einzelne Zellen. Eine Zelle stellt daher immer eine Sonderform eines Bereiches dar – einen einzelligen Bereich. Zu beachten ist außerdem, dass viele auf Bereiche anwendbare Befehle nur mit einzelligen Bereichen umgehen können.

5.1.2. Zugriff auf Zellen und Zellbereiche

Die Bearbeitung von Bereichen erfolgt entweder direkt durch Angabe des Bereiches, auf den zugegriffen werden soll oder über bereits markierte Zellbereiche. Bereichsobjekte werden dabei mit den Methoden ***Select*** und ***Activate*** markiert und damit ausgewählt. Ist der Bereich ausgewählt kann man mit den Methoden ***ActiveCell***, ***Selection*** oder ***UsedRange*** auf die Bereiche zugreifen. *ActiveCell* meint dabei die gerade markierte, aktive Zelle, *Selection* liefert den markierten Bereich, und *UsedRange* gibt denjenigen Bereich des Tabellenblattes als Bereichsobjekt zurück, der mit Inhalt gefüllte Zellen enthält.

Möchte man auf noch nicht markierte Bereiche zugreifen, so benutzt man die Range-Methode, die in diesem Zusammenhang inhaltlich vom Range-Objekt abzugrenzen ist. Die Range-Methode liefert lediglich ein Range-Objekt zurück. Das folgende Beispiel soll die Syntax verdeutlichen: `Range("A1:B4")` oder `Range("A2")`. Das erste Beispiel liefert den Zellbereich als Range-Objekt zurück, der von den Zellen A1 bis B4 aufgespannt wird. Im zweiten Beispiel liefert die Range-Methode ein einzelliges Range-Objekt zurück, das auf Zelle A2 verweist. Es ist ebenfalls folgende Kurzschreibweise möglich, bei der der Range-Befehl durch eckige Klammern ersetzt wird: `Range[A1:B4]` oder `[A2]`.

Mit den Methoden ***ClearFormats***, ***ClearContents*** und ***Clear*** des Bereichobjektes kann der Zellinhalt bearbeitet werden. *ClearFormats* löscht dabei nur die Formatierungen der Zellen des Bereichs, *ClearContents* nur die Inhalt, die Formatierungen bleiben erhalten. Die Methode *Clear* löscht sowohl die Formatierungen als auch den Inhalt.

Mit den Methoden ***Insert*** und **Delete** des Bereichsobjektes können Zellbereiche eingefügt bzw. entfernt werden.

Neben der *Range*-Methode erlaubt die *Cells*-Methode komfortableren Zugriff auf einzellige Bereiche eines Tabellenblatts oder eines Bereiches. Cells kann auf zwei verschieden Art und Weisen verwendet werden. ***Cells(m,n)*** liefert die Zelle in Zeile m und Spalte n als Range-Objekt zurück. ***Cells(n)*** liefert die n-te Zelle, wobei alle Zellen zeilenweise durchnummeriert werden. Zelle A1 hat den Index 1, Zelle B1 den Index 2 usw.

Eine weitere wichtige Bereichsobjekt-Methode ist ***Offset***. *Offset(m,n)* liefert den um m Zeilen und n Spalten versetzten Bereich des aktuell markierten Bereichs wieder.

Zusätzlich zu diesen Methoden besitzt das Range-Objekt noch eine Reihen wichtiger Eigenschaften, um auf den Inhalt von einzelligen Bereichen zuzugreifen. Einige dieser Eigenschaften sollen kurz aufgezählt werden. Die Online-Hilfe bietet genauere Informationen. Kurze Aufzählung: *Value, Text, Characters, FormulaLocal, FormulaR1C1Local, Formula, FormulaR1C1, HasFormula, Font, Orientation, NumberFormat, Style, Borders, BorderAround.*

Zum Abschluss dieses Abschnitts sollen einige Beispiel Aufschluss über die Verwendung der angesprochenen Methoden liefern:

```
Cells(4,6).Activate          'markiert Zelle F4
Range("A1:B3").Cells(4)      'liefert Zelle B2
[B2].Offset(1,3).Select      'markiert Zelle E3
Selection.Cells(1).Text = "Inhalt"
[A2].Value = 12
[A2].Font.Bold = True
```

5.2. Arbeitsmappen, Fenster, Arbeitsblätter

Eine Auflistung von Methoden und Eigenschaften ist im Anhang zu finden. Der Inhalt diese Kapitels wird später fertiggestellt.

5.2.1. Zugriff auf Arbeitsmappen, Fenster, Arbeitsblätter

5.2.2. Umgang mit Arbeitsmappen

5.2.3. Umgang mit Fenstern

5.2.4. Umgang mit Arbeitsblättern

A. Syntaxzusammenfassung und -überblick

A.1. Zugriff auf/Umgang mit Arbeitsmappen, Fenster und Blätter

Zugriff auf Arbeitsmappen, Fenstern und Blättern

Workbooks	Aufzählungsobjekt – Zugriff auf alle Arbeitsmappen
Windows	Aufzählungsobjekt – Zugriff auf alle Fenster
Sheets	Zugriff auf alle Arbeitsblätter einer Mappe
Worksheets	Zugriff nur auf Tabellenblätter
Charts	Zugriff nur auf Diagrammblätter
ActiveWorkbook	zur Zeit aktive Arbeitsmappe
ThisWorkbook	Arbeitsmappe, in der sich der aktuelle Code befindet
ActiveWindow	aktives Fenster
ActiveSheet	aktives Arbeitsblatt
ActiveChart	aktives Diagrammblatt

Umgang mit Arbeitsmappen

mappe.Activate	bestimmt die aktive Arbeitsmappe
Workbooks.Add	erstellt eine neue leere Arbeitsmappe
mappe.Close	schließt die Arbeitsmappe
mappe.Open “dateiname”	lädt die angegebene Datei
mappe.Save	speichert die Arbeitsmappe
mappe.SaveAs “dateiname”	speichert die Arbeitsmappe unter angegebenem Namen
mappe.Name	enthält den Dateinamen ohne Pfad
mappe.Path	gibt nur den Pfad der Arbeitsmappe an
mappe.FullName	Pfad und Dateiname
mappe.Saved	gibt an, ob die Arbeitsmappe gespeichert ist

Umgang mit Fenstern

fenster.Activate	aktiviert das angegebene Fenster
fenster.ActivatePrevious	aktiviert das zuletzt aktive Fenster
fenster.ActivateNext	aktiviert das nächste Fenster der Fensterliste
fenster.Close	schließt das angegebene Fenster
fenster.NewWindow	erzeugt eine neues Fenster
fenster.Visible	Fenster ein-/ausgeblendet
fenster.Caption	gibt den Fenstertitel an
fenster.DisplayGridlines	Gitter anzeigen
fenster.DisplayHeadings	Zeilen- und Spaltenköpfe anzeigen
fenster.Zoom	Zoomfaktor einstellen (10-400)
fenster.ScrollColumn	sichtbare Spaltennummer am linken Rand
fenster.ScrollRow	sichtbare Zeilennummer am oberen Rand
fenster.Split	gibt an, ob das Fenster geteilt ist
fenster.SplitRow	Zeilenzahl im oberen Fensterteil bei Teilung
fenster.SplitColumn	Spaltenzahl im linken Fensterteil bei Teilung
fenster.Width	Breite des Fensters in Punkt (0.35 mm)
fenster.Height	Höhe des Fensters in Punkt (0.35 mm)

Umgang mit Blättern

blatt.Activate	wählt ein Blatt aus
mappe.Add	fügt ein leeres Tabellenblatt hinzu
blatt.Copy	kopiert das Blatt in eine neue Mappe
blatt.Delete	löscht das Blatt (mit Sicherheitsabfrage)
blatt.Name	Name des Blattes
blatt.Visible	Blatt ein- oder ausgeblendet

A.2. Umgang mit Zellen und Zellbereichen

Zugriff auf ausgewählte Bereiche

ActiveCell	aktive Zelle
Selection	markierter Bereich oder markiertes Objekt im Fenster
UsedRange	genutzter Bereich in Tabellenblatt

Zugriff auf beliebige Bereiche

Range(“A2”)	eine Zelle
Range(“A2:B4”)	Zellbereich
Range(“name”)	Zugriff auf einen benannten Bereich
[A2], [A2:B4], [name]	Kurzschreibweise für Range-Methode
Range(bereich1,bereich2)	Bereich zwischen Zellen *bereich1* und *bereich2*; *bereich1* und *bereich2* können auch durch *Cells* angegeben werden
bereich.Offset(n,m)	liefert den um n Zeilen und m Spalten versetzten Bereich ausgehend von *bereich*
bereich.Resize(n,m)	vergrößert den Bereich *bereich* um n Zeilen und m Spalten
bereich.Select	wählt den angegebenen Bereich aus
bereich.Activate	wählt den angegebenen Bereich aus

Zugriff auf spezielle Zellen

bereich.Cells	Aufzählobjekt aller Zellen
bereich.Cells(n)	Zugriff auf n-te Zelle bei zeilenweiser Durchnummerierung (1=A1, 2=B1,..., 257=A2, ...)
bereich.Cells(n,m)	Zelle der n-ten Zeile und m-ten Spalte
bereich.Areas	Aufzählobjekt aller rechteckigen Bereiche
bereich.Areas(n)	n-ter rechteckiger Bereich
bereich.EntireColumn	Spalten, in denen sich der Bereich befindet
bereich.EntireRow	Zeilen, in denn sich der Bereich befindet
bereich.Columns(n)	Zugriff auf n-te Spalten des Bereiches
bereich.Rows(n)	Zugriff auf n-te Zeile des Bereiches
bereich.Address(...)	liefert eine Zeichenkette mit der Bereichsadresse

Zellbereich einfügen/löschen

bereich.ClearContents	Zellinhalte löschen
bereich.ClearFormats	Zellformatierung löschen, Inhalt bleibt bestehen
bereich.Clear	Inhalte und Formate des Bereiches löschen
bereich.Delete	Zellen löschen
bereich.Insert	Zellen einfügen

Inhalt und Format von Zellen

zelle.Value	Inhalt der Zelle, bei Formeln wird das Ergebnis geliefert
zelle.Text	formatierte Zeichenkette mit Inhalt der Zelle (Nur-Lesen)
zelle.Characters(n,m)	liefert ab dem n-ten Zeichen m Zeichen des des Textes der Zelle
zelle.Formula	Formel der Zelle in A1-Schreibweise
zelle.FormulaR1C1	Formel der Zelle in R1C1-Schreibweise
zelle.HasFormula	gibt an, ob die Zelle eine Formel enthält
zelle.Font	Verweis auf das Schriftarten-Objekt
zelle.VerticalAlignment	vertikale Ausrichtung des Zellinhalts
zelle.HorizontalAlignment	horizontale Ausrichtung des Zellinhalts
zelle.Orientation	Textrichtung (horizontal/vertikal)
zelle.WrapText	Zeilenumbruch in der Zelle
zelle.ColumnWidth	Breite der ganzen Spalte
zelle.RowHeight	Höhe der ganzen Zeile
zelle.NumberFormat	Zahlenformat der Zeichenkette
zelle.Style	gibt ein Style-Objekt zurück, das die Formatvorlage der Zelle darstellt
zelle.BorderAround	zur Einstellung des Gesamtrahmen der Zelle
zelle.Row	Zeilennummer der Zelle
zelle.Column	Spaltennummer der Zelle

A.3. Sonstige erwähnenswerte Funktionen

Funktionen für Zeichenketten

Left(z,n)	liefert die ersten n Zeichen der Zeichenkette z
Right(z,n)	liefert die letzten n Zeichen der Zeichenkette z
Mid(z,n)	liefert alle Zeichen ab dem n-ten
Mid(z,n1,n2)	liefert $n2$ Zeichen ab dem $n1$-ten Zeichen
Mid(z1,n1,n2) = z2	setzt $z2$ in $z1$ ein
Len(z)	ermittelt die Länge der Zeichenkette
InStr(z1,z2)	sucht String z2 in z1; Ergebnis: Position oder 0
UCase(z)	wandelt alle Klein- in Großbuchstaben
LCase(z)	wandelt alle Groß- in Kleinbuchstaben
Trim(z)	löscht alle Leerzeichen am Anfang und am Ende der Zeichenkette
String(n,"x")	liefert eine Zeichenkette aus n mal "x"
Space(n)	liefert eine Zeichenkette bestehend aus n Leerzeichen
MsgBox "text"	zeigt einen Meldungsfenster mit Text "text"
MsgBox("text", buttons)	zeigt einen Meldungsdialog mit Auswahltasten an
InputBox("text")	zeigt eine Eingabedialog zur Abfrage einer Zeichenkette an

Funktionen zum Runden

Int(f)	rundet immer ab
Fix(f)	scheidet die Nachkommastellen ab
Round(f,n)	rundet bei 0.5 auf n Stellen
RoundDown(f,n)	rundet immer ab mit n Nachkommastellen
RoundUp(f,n)	rundet immer auf mit n Nachkommastellen
Even(f)	rundet zur betragsmäßig größeren geraden Zahl
Odd(f)	rundet zur betragsmäßig größeren ungeraden Zahl
Ceiling(f1,f2)	rundet zum Vielfachen von $f2$ auf
Floor(f1,f2)	rundet zum Vielfachen von $f2$ ab

Sonstige numerische Funktionen

Abs(z)	liefert den Absolutbetrag
Sqn(z)	Signum-Funktionen
Sqr(z)	liefert die Quadratwurzel
Sin(z),Cos(z),Tan(z)	trigonometrische Funktionen
Atn(z)	Arcustangens-Funktion
Log(z),Exp(z)	logarithmische Funktionen
Rnd	liefert eine Zufallszahl zwischen 0 und 1
Randomize	initialisiert den Zufallszahlengenerator neu