Tutorial

1

EDV INNOVATION & CONSULTING

Stefan Kulpa

Gerhard-Domagk-Str. 6
D-41540 Dormagen
http://www.kulpa-online.com
Email: stefan@kulpa-online.com

VBA Einführung

LEVEL BEGINNER

Inhaltsverzeichnis

Einführung ... 3

Was ist Visual Basic für Applikationen? ... 3
Was ist Visual Basic? ... 3
Worin unterscheiden sich Visual Basic, VBA und VBScript? ... 3
Worin unterscheiden sich VBA und Visual Basic? ... 4
VBA ist nicht BASIC... ... 4
Die integrierte Entwicklungsumgebung ... 4
Objektorientierte Programmierung mit VBA ... 6
Was ist ein Objekt ... 6
Objekte als Komponenten von VBA-Anwendungen ... 6

Ein VBA-Programm erstellen ... 8

Schritt 1: Ein Beispiel-Programm entwerfen ... 8
Schritt 2: Das Design implementieren ... 8
Eine neue UserForm hinzufügen ... 9
Der UserForm ein Bezeichnungsfeld hinzufügen ... 10
Das Eigenschaften-Fenster benutzen ... 10
Eine Befehlsschaltfläche hinzufügen ... 12
Schritt 3: Den Code schreiben ... 13
Das Codefenster des Formulars öffnen ... 13
Eigenen Code hinzufügen ... 14
Eine zweite Prozedur erstellen ... 15
Den Code für die neue Prozedur schreiben ... 16

Ein VBA-Programm ändern ... 17

Anpassen des Formulars ... 17
Plausibilisierung der Postleitzahl ... 19
Plausibilisierung der Texteingaben ... 22
Plausibilisierung des Geburtsdatums ... 23
Alter berechnen ... 25
Musterlösung ... 30

Tipps & Tricks – Routinen ... 33

Subs und Functions oder die Frage: Wie finde ich die richtigen Argumente? ... 33
Sub-Routinen geben keinen Wert zurück, Function Routinen können dies jedoch. Richtig? Jain! ... 34
Was ist eine Property oder: Wie schütze ich meine Variablen? ... 35
Wo liegt denn jetzt der Vorteil? ... 37
Nur Wiederholungen mit Schleifen ... 38
Do-Schleifen ... 39
Kopf oder Fuß – die Bedingung diktiert den Ablauf ... 40
Wann man Do ohne While und Until verwendet ... 40
Abzählen mit For...Next-Schleifen ... 41
Verschachtelte For...Next Schleifen ... 42

Abbildungsverzeichnis ... 43

Kapitel

1

EINFÜHRUNG

Was ist Visual Basic für Applikationen?

Microsoft **Visual Basic für Applikationen** (VBA) ist Teil der Technologiefamilie **Visual Basic**, zu der das Entwicklungssystem Microsoft Visual Basic (Learning Edition, Professional Edition und Enterprise Edition) sowie Visual Basic Scripting Edition (VBScript) gehören.

VBA ist eine integrationsfähige Programmier-Umgebung. Sie soll Programmierer in die Lage versetzen, auf Basis der gesamten Leistungsstärke von Visual Basic aus Standardanwendungspaketen maßgeschneiderte Lösungen zu entwickeln. Programmierer, die mit Anwendungen arbeiten, die Visual Basic für Applikationen beinhalten, können die Funktionalität dieser Anwendungen (die dem Programmierer in Form einer Objektmenge offen gelegt wird) automatisieren, erweitern und integrieren, wodurch der Entwicklungszeitraum maßgeschneiderter Branchenlösungen verkürzt wird.

Was ist Visual Basic?

Das Microsoft Visual Basic-**Entwicklungssystem** ist das produktive Tool zur Erstellung hocheffizienter Komponenten und Anwendungen. Mit Visual Basic können Entwickler robuste Anwendungen erstellen, die auf dem Client oder Server residieren oder in einer verteilten mehrschichtigen Umgebung ausgeführt werden. Das **RAD-Tool** Visual Basic steht entweder als eigenständiges Produkt oder als Teil des Visual Studio-Tool-Pakets zur Verfügung.

Worin unterscheiden sich Visual Basic, VBA und VBScript?

Visual Basic ist ein **eigenständiges** Werkzeug zur Erstellung voneinander unabhängiger Softwarekomponenten, wie z. B. ausführbaren Programmen, **COM-Bestandteile** und **ActiveX-Steuerelementen**. Es eignet sich für Speziallösungen, die von Grund auf neu erstellt werden sollen. VBA bietet im Kontext einer vorhandenen Anwendung die gleichen leistungsfähigen Werkzeuge wie Visual Basic und stellt die beste Wahl für die Anpassung bereits vorhandener Software dar. **VBScript** ist eine Variante der Sprache Visual Basic, die speziell für den Einsatz in Webseiten konzipiert wurde. Obgleich VBScript gelegentlich für einfache Automatisierungsaufgaben verwendet werden kann, ist VBA die Technologie, die speziell für die **Anwendungsautomatisierung** konzipiert wurde.

Worin unterscheiden sich VBA und Visual Basic?

Visual Basic ist ein **unabhängiges** Werkzeug für die rasche Erstellung ausführbarer Programme und Softwarekomponenten. VBA ist eine integrationsfähige Programmier-Umgebung zur Automatisierung und Anpassung der Software, in die sie eingebettet ist.

VBA ist nicht BASIC...

Man könnte sagen, dass VBA ein direkter Nachfahre der Original-BASIC-Programmiersprache ist. Aber das wäre genau so, wie wenn man behaupten würde, dass Menschen von Amöben abstammen. In beiden Fällen hat dazwischen eine umfangreiche Entwicklung stattgefunden.

Die Originalversion von BASIC (**Beginner's All-purpose Symbolic Instruction Code**) wurde in den 1960er Jahren geschaffen in der Hoffnung, Programmierung für den Normalanwender einfacher zu machen. Verglichen mit anderen Programmiersprachen wie C++ oder Fortran sind die BASIC-Befehle mehr wie normales Englisch; sie sind einfach zu verstehen und zu merken.

Viele der **„Wörter“** für besondere Zwecke, die in der VBA-Programmiersprache benutzt werden, waren Teil der Original-BASIC-Sprache und hatten ähnliche Funktionen. So ist es auch mit der VBA-„Grammatik“: Einige der Regeln dafür, welche Worte kombiniert werden können und in welcher Reihenfolge, stammen aus BASIC.

Die **VBA-Sprache** hat sich allerdings wesentlich weiter entwickelt als den früheren BASIC-Versionen. Viele der **VBA-Befehle** und die Regeln dafür, wie man sie kombinieren kann, waren nicht Teil des Original-BASIC. Die VBA-Sprache kann mehr als das alte BASIC, viel mehr sogar, speziell im Hinblick auf die Darstellung interessanter Formen auf dem Bildschirm und die Interaktion mit der anderen Software.

Außerdem ist VBA nicht nur eine Programmiersprache. VBA enthält eine komplette Software-Entwicklungsumgebung (**IDE = Integrated Development Environment**) mit vielerlei Fenstern für spezielle Zwecke, die einem helfen, eigene Programme zu gestalten und zu testen.

Und VBA hat etwas, wovon man bei BASIC nicht einmal träumen konnte – mit VBA kann man eine anspruchsvolle Benutzeroberfläche erstellen, ohne Programmcode per Hand schreiben zu müssen. Diese Funktion macht VBA zu einem visuellen Software-Entwicklungstool.

Die integrierte Entwicklungsumgebung

Die gesamte Arbeit mit VBA findet in der integrierten Entwicklungsumgebung statt (**IDE**). Dieser Begriff mag kalt und Furcht erregend klingen, aber man sollte sich nicht einschüchtern lassen – man sollte sich diese Umgebung einfach als eine gemütliche Werkstatt vorstellen, in der man alle seine Programmieraufgaben bequem erledigen kann.

Die VBA-Entwicklungsumgebung wird als **Visual Basic-Editor** genannt. Der Visual Basic-Editor besteht aus einem Top-Level-Anwendungsfenster mit einem Menü- und Symbolleistensystem, dass einem alle Werkzeuge zur Verfügung stellt, die man braucht, um seine Programme zu entwickeln.

Die nachfolgende Abbildung zeigt die Word-Version des Visual Basic-Editors. Der Editor sieht aber in jeder Anwendung gleich aus.

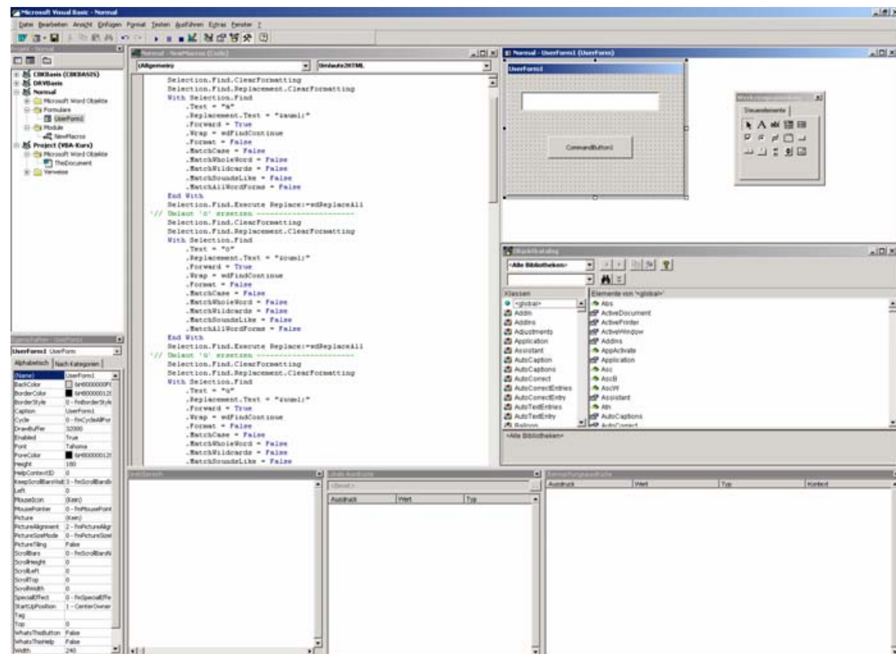


Abbildung 1 - Die DIE

Der Visual Basic-Editor wird aus der **Office Anwendung** (Word, Excel etc.) über das Menü Extras – Makro – Visual Basic Editor oder über die Tastenkombination **ALT + F11** aufgerufen.

Der erste Anblick des Visual Basic-Editors mag verwirrend sein. Die Menüs und Symbolleisten oben auf dem Bildschirm kommen einem sicher bekannt vor, aber was ist mit all den Fenstern? Da den Überblick zu behalten, ist gar nicht so leicht, denn sie können in unendlich vielen Varianten angeordnet und kombiniert werden.

Eine kurze Zusammenfassung der Namen und Funktionen kann der folgenden Tabelle entnommen werden:

Art des Fensters	Wozu es nützlich ist
Projekt-Explorer-Fenster	Verwaltung der Komponenten eines VBA-Projekts
Codefenster	Anzeige und Bearbeitung des VBA-Codes
UserForm-Fenster	Entwicklung von Standard-Formularen (Fenster und Dialogfelder)
Werkzeugsammlung	Steuerelemente wie zum Beispiel Textfelder und Schaltflächen hinzufügen, in manchen VBA-Anwendungen direkt in Dokumente, ansonsten in Formulare

Art des Fensters	Wozu es nützlich ist
Eigenschaften-Fenster	Individuelle Attribute ausgewählter Formulare oder Steuerelemente festlegen
Überwachungsfenster	Die Werte ausgewählter Programmvariablen des Programms verfolgen
Lokal-Fenster	Die Werte der Variablen der aktuellen Prozedur verfolgen
Direktfenster	Individuelle Zeilen von Programmcode testen, um sofortige Ergebnisse zu erzielen
Objektkatalog	Die für das Programmfenster verfügbaren Objekte herausfinden

Objektorientierte Programmierung mit VBA

Visual Basic hat sich zu einer (fast) vollständigen objektorientierten Programmiersprache entwickelt. Wenngleich die Arbeit mit Objekten eine Herausforderung für den VBA-Neuling darstellt, die Ergebnisse ist jede Anstrengung wert.

Was ist ein Objekt

Durch Objekte – lebendige Funktionen der VBA-Landschaft – hat man Zugang zur Funktionalität der zugrunde liegenden VBA-Anwendung, in der man arbeitet (Word, Excel etc.). Darüber hinaus kann man noch auf Objekte aus anderen kompatiblen Anwendungen zugreifen und sogar eigene Objekte schaffen.

Vom Standpunkt der Praxis aus gesehen, ist ein Objekt einfach ein benanntes Element: Es hat:

- **Eigenschaften:** Einstellungen, die überprüft und geändert werden können
- **Methoden:** Aktionen, die das Objekt durchführen kann, wenn das Programm es dazu auffordert
- **Ereignisse:** Dinge, die mit dem Objekt passieren, auf welche es reagieren kann, indem es automatisch eine vorher definierte Aktion durchführt

Etwas ist ein Objekt, wenn es sowohl Daten als ach Code, der die Daten bearbeitet, besitzt. Ein Objekt schließt die Daten und den dazugehörigen Code ein.

Objekte als Komponenten von VBA-Anwendungen

Der einfachste Weg, sich Objekten anzunähern, ist, sie sich als Teile der eigenen VBA-Anwendung und Dokumente vorzustellen. Eine **Form** in einer Visio-Zeichnung ist ein Objekt, genau wie jede Verbindungs(linie), die zwei Formen verbindet. Das trifft auch für jede Ebene, der man Formen zuordnen kann, und für jede Seite zu, zu dem alle Seiten, Ebenen, Formen und Linien gehören.

Genauso ist es in Excel. Zu Excel-Objekten gehören die Zellen, in die man Daten und Formeln eingibt, bestimmte Bereiche von Zellen, die Diagramme, die viele Arbeitsblätter illustrieren,

einzelne Arbeitsblätter sowie komplette Arbeitsmappen. Und in den meisten VBA-Anwendungen sind die Symbolleistungen und Menüs, die Schaltflächen und Auswahlelemente, die sie erhalten, ebenfalls Objekte. VBA-Objekte existieren in einer Hierarchie, in der eine Art von Objekt andere Arten von Objekten enthält. Diese Objekt-Hierarchien werden **Objektmodelle** genannt.

Kapitel 2

EIN VBA-PROGRAMM ERSTELLEN

Schritt 1: Ein Beispiel-Programm entwerfen

Das Beispielprogramm öffnet ein neues Fenster, in dem es einen Hinweis mit Datum und Uhrzeit anzeigt (nachfolgend die „Nachricht" genannt). Das Fenster bleibt geöffnet, bis der Nutzer auf OK klickt.

Ausgehend von dieser Beschreibung kann man eine Liste der Elemente aufstellen, die für dieses Programm notwendig sind:

- Das Programm hat offensichtlich ein Fenster. Man braucht daher ein Formular (**UserForm**).
- Auf dem Formular werden zwei **Steuerelemente** gebraucht: ein **Bezeichnungsfeld** für die Nachricht und eine **Schaltfläche** für das **OK**.
- Man muss auch **Code** für zwei Prozeduren schreiben: eine, die die Nachricht im Bezeichnungsfeld anzeigt und eine andere, die dafür sorgt, dass das Programm geschlossen wird, wenn der Nutzer auf OK klickt.

Was dieses Programm nicht benötigt, ist ein separates Modul für den **VBA-Code** (ein **Modul** ist eine **Code-Einheit**, die eine oder **mehrere Prozeduren** enthält).

Man kann beide Prozeduren in dem **Codefenster** schreiben, das dem Formular selbst zugeordnet ist. Denn beide Prozeduren tun ihre Arbeit als Antwort auf **Ereignisse**, die im Formular selbst geschehen.

Betrachten wir einmal die Prozedur, die Nachricht und Datum anzeigt: diese benötigen wir erst in dem Moment, in dem das Programmfenster auf dem Bildschirm erscheint. In diesem Programm passiert das schneller, als man mit den Augen zwinkern kann. Die Anzeige des Formulars ist das Ereignis, das diese Prozedur auslöst.

Die Prozedur zum Verlassen des Programms wird ausgelöst von einem Ereignis, das mit dem Formular in Zusammenhang steht. In diesem Fall ist es die Tatsache, dass der Nutzer auf OK klickt. Auch diese Prozedur kann man im Codefenster des Formulars schreiben.

Schritt 2: Das Design implementieren

Wenn man eine klare Vorstellung hat von dem, was man tun will, kann man ernsthaft anfangen zu programmieren.

Eine neue UserForm hinzufügen

Nachdem der Visual Basic-Editor aufgerufen wurde, wählt man **„EINFÜGEN – USERFORM“** aus der Menüleiste des Visual Basic-Editors, um ein neues Formular auf dem Bildschirm zu platzieren.

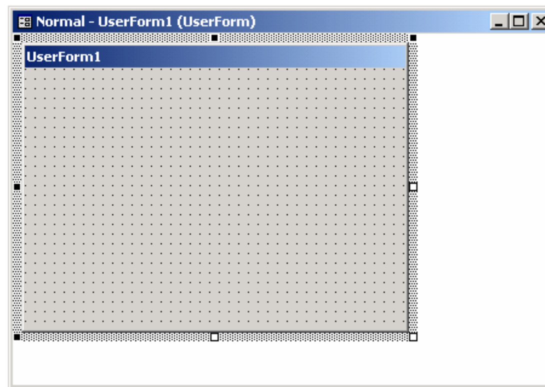


Abbildung 2 - Eine UserForm

Wie man in der Abbildung sehen kann, ist die neue UserForm eine funktionslose graue Ebene. Hier ist zu beachten, dass man die Größe der UserForm durch Ziehen an den kleinen **weißen Vierecken** an der rechten und der unteren Seite beeinflussen kann (das sind die **Ziehpunkte**).

Das Fenster für dieses Programm sollte ein wenig breiter sein (damit der gesamte Text der Nachricht in eine Zeile passt) und ein wenig kürzer (eine Zeile Text und eine Schaltfläche nehmen nicht soviel Platz weg).

Noch interessanter ist die **Werkzeugsammlung** – die Palette mit den vielen Icons, die erscheinen, wenn man mit einem Formular arbeitet. Man benutzt diese Werkzeugsammlung, um Steuerelemente auf den Formularen zu platzieren.

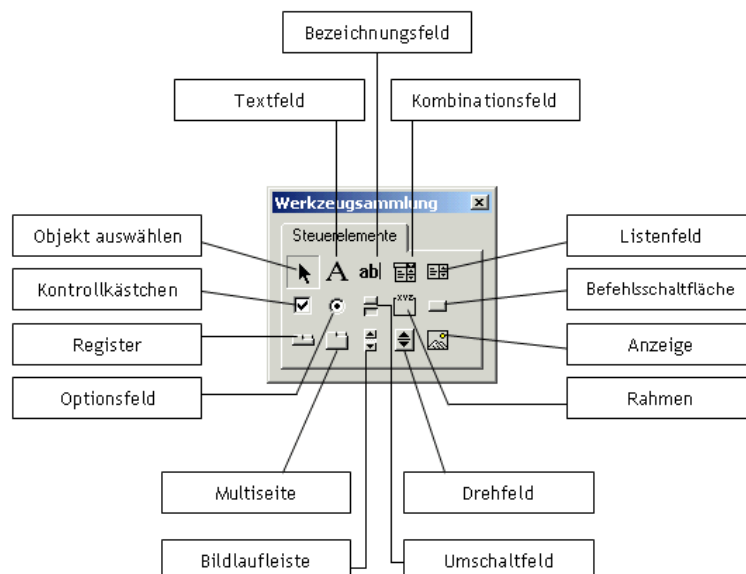


Abbildung 3 - Die Werkzeugsammlung

Die **VBA-Toolbox**, gut gefüllt mit Steuerelementen, die man seinen Formularen hinzufügen kann.

Der UserForm ein Bezeichnungsfeld hinzufügen

Nun sind wir soweit, dass dem Formular Leben eingehaucht werden kann, indem Steuerelemente hinzugefügt werden. Beginnen wir mit einem **Bezeichnungsfeld**, welches einfach nur Text anzeigt, die Überschrift also. Wenn das Programm läuft, ist das Bezeichnungsfeld etwas, das man sich lediglich ansehen kann. Die Nutzer können den Text lesen, aber nicht verändern.

Das Programm kann das aber. Unser Beispielprogramm braucht diese Fähigkeit, da es bei jedem Programmaufruf jeweils das aktuelle Datum und die Uhrzeit anzeigen soll.

Um in die neue UserForm ein Bezeichnungsfeld einzufügen, sind folgende Schritte zu durchzuführen:

1. Das Formularfenster muss aktiv sein; dies lässt sich sicherstellen, in dem man darauf klickt.
2. Die Werkzeugsammlung ist nur sichtbar, wenn das Formular aktiv ist.
3. Klicke in der Werkzeugsammlung auf das Icon mit dem großen A.
4. Bewege den Cursor nun zur UserForm und ziehe, beginnend auf der linken Seite oben, ein Rechteck, das groß genug ist, dass die Nachricht darin Platz hat.

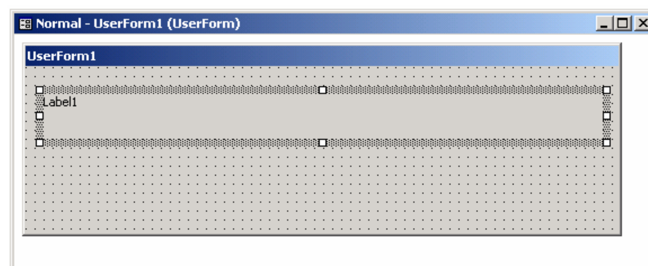


Abbildung 4 - Ein Bezeichnungsfeld einfügen

VBA vergibt jedem neuen Bezeichnungsfeld automatisch eine Überschrift, so dass man an diesem Punkt die viel sagende Überschrift **Label1** innerhalb des rechteckigen Bezeichnungsfeldes sehen sollte. Das ist noch nicht so ganz das Wahre. Um die automatische Überschrift zu löschen, braucht man das **Eigenschaften-Fenster**.

Das Eigenschaften-Fenster benutzen

Jedes Steuerelement auf einem Formular hat eine lange Liste von **Eigenschaften**. Sie legen fest, wie das Steuerelement aussieht und wie es sich verhalten soll, wenn das Programm läuft.

Eines der großen Stärken von VBA ist, dass man diese Eigenschaften kontrollieren kann, ohne auch nur eine einzige Zeile Code schreiben zu müssen.

Das Steuerfeld für Eigenschaften ist das **Eigenschaften-Fenster**. Wie man in der nachfolgenden Abbildung sieht, enthält es eine Liste aller Eigenschaften des Steuerelements, das gerade ausgewählt ist. Um eine Eigenschaft zu ändern, muss man lediglich die Eigenschaft aus der linken Spalte heraussuchen und die dazugehörige Einstellung in der rechten Spalte ändern.

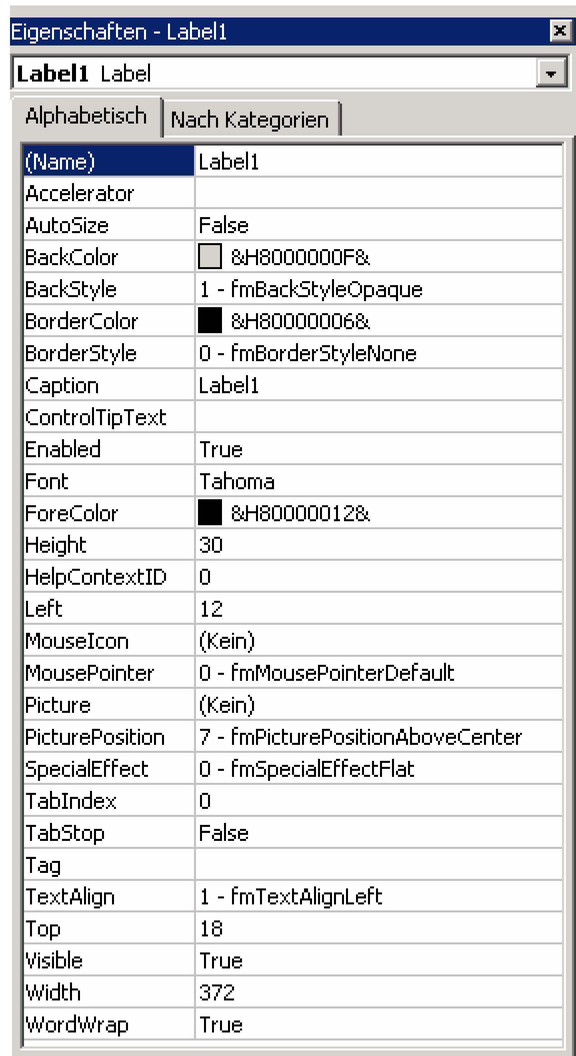


Abbildung 5 - Das Eigenschaften-Fenster

Im Falle des Bezeichnungsfeldes in unserem Beispielprogramm, müssen genau zwei Eigenschaften geändert werden: **Name** und **Caption**. Ändere den Namen des Steuerelements von **Label1** zum Beispiel in **lblNow** (anhand der ersten drei Buchstaben kann man feststellen, dass es sich hier um ein Bezeichnungsfeld handelt: lbl = Label = Bezeichnungsfeld). Klicke dazu auf das Bezeichnungsfeld, um es auszuwählen. Die Eigenschaft **Name** findet man ganz oben auf der Eigenschaften-Liste, dort erscheint sie so: (Name). Es ist die einzige Eigenschaft in Klammern. Doppelklicke in der rechten Spalte dieser Reihe, so dass der bereits vorhandene Name markiert wird, und gib dann den neuen Namen ein.

Die Namen der Steuerelemente benutzt man, um sich auf das Steuerelement zu beziehen, wenn man Code schreibt. Aus dem Grund sollte man versuchen, Namen zu wählen, die eine Beziehung zum Namen des Steuerelements haben.

Weiter unten in der Eigenschaften-Liste wird man das Feld **Caption** finden. Hier ist einfach der aktuelle Eintrag zu löschen. Warum? Weil das Programm für den entsprechenden Überschriften-Text sorgen wird, wenn es erst einmal läuft.

Solange man dabei ist, die Eigenschaften zu ändern, kann man auch das Formular selbst umbenennen. Das Formular wird ausgewählt, indem man auf die Titelleiste des UserForm-Fensters klickt. Nun kann man im Eigenschaften-Fenster die Eigenschaft Caption heraussuchen und einen entsprechenden Text eingeben(z.B.: „Mein erstes Programm“). Die neue Überschrift erscheint nun in der Titelleiste des Formulars.

Eine Befehlsschaltfläche hinzufügen

Im Gegensatz zu einem Bezeichnungsfeld kann eine Befehlsschaltfläche (oder auch Schaltfläche) mit dem Nutzer interagieren. Wenn jemand auf die Schaltfläche klickt, sieht sie eingedrückt aus und das Programm tut etwas.

Unser Beispielprogramm braucht nur eine Schaltfläche, und zwar die, die das Programm beendet, wenn der Nutzer darauf klickt. Gehe folgendermaßen vor, um die Schaltfläche auf dem Formular zu platzieren:

1. Klicke auf das Formularfenster, um es noch einmal zu aktivieren.
2. Klicke in der Werkzeugsammlung auf das Icon **Befehlsschaltfläche**.
3. Ziehe ein Rechteck. Fange dabei in der linken oberen Ecke der zu zeichnenden Schaltfläche an und ziehe diagonal nach rechts unten.

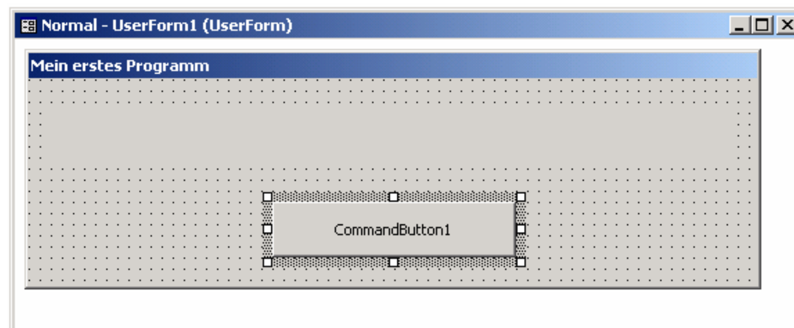


Abbildung 6 - Die Befehlsschaltfläche hinzufügen

Jetzt können wir die Eigenschaften für die Schaltfläche festlegen.

1. Ändere die Eigenschaft **(Name)** in **OKButton**
2. Ändere den Wert im Feld **Caption** in **OK**

Das Formular für das Beispielprogramm ist fertig. Auch ohne weitere Arbeit haben wir bereits ein funktionierendes VBA-Programm, und zwar eines, welches das unwahrscheinlich faszinierende Formular anzeigt, dass wir gerade entworfen haben. Benutze die Taste **F5**, um es auszuführen.

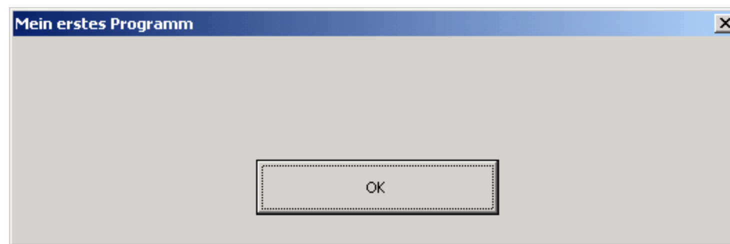


Abbildung 7 - Das bisherige Ergebnis

Wenn wir das kleine, halbfertige Programm ausführen, erscheint das Formular in der Office-Anwendung (nicht im Visual Basic-Editor). Das Bezeichnungsfeld ist leer und die Schaltfläche OK noch ohne Funktion.

Um das Programm zu schließen, müssen wir die Standardschaltfläche betätigen, die Windows dafür vorgesehen hat, nämlich das kleine X am rechten Ende der Titelleiste.

Schritt 3: Den Code schreiben

Nun kommt der aufregendste Teil der VBA-Programmierung: Das Schreiben des Codes, der dafür sorgt, dass unser Programm läuft.

Das Codefenster des Formulars öffnen

Um den Code zu schreiben, der hinter einem Formular oder hinter einem seiner Steuerelemente steht (mit ihnen verbunden ist), müssen wir statt des Formulars das Codefenster anzeigen. Dazu muss das Formular oder Steuerelement zunächst einmal ausgewählt werden. Die Prozedur für die Schaltfläche OK ist einfacher, lass und also damit anfangen. Klicke auf die Schaltfläche, so dass an deren Ecken die Ziehpunkte erscheinen. Mit einer der nun folgenden Methoden können wir das Codefenster anzeigen lassen:

- Wähle **ANSICHT – CODE**.
- Betätige die Taste **F7**.
- Klicke das Steuerelement mit der rechten Maustaste an und wähle **CODE ANZEIGEN** aus dem Kontextmenü aus.

Wenn das Codefenster erscheint, sollte es bereits das Grundgerüst einer Prozedur enthalten. VBA erstellt bereits automatisch eine Prozedur für die allgemeinste Art von Ereignis, nämlich den einfachen Mausklick auf die Schaltfläche. Das Programm führt diese Prozedur aus, wann immer jemand auf die Schaltfläche OK klickt.

Die zwei Codezeilen, die man im Prozedur-Grundgerüst sieht, bewirken noch nichts. Es sind einfach nur Grenzlinien, die VBA sagen, wo eine Prozedur anfängt und endet. Die erste Codezeile, die VBA generiert hat, lautet:

```
Private Sub OKButton_Click()
```

In allen VBA-Prozeduren definiert die erste Codezeile die Art der Prozedur (in diesem Fall eine „private“ **Sub**-Routine) und ihren Namen. **Private** und **Sub** sind VBA-**Schlüsselwörter**, Wörter oder Symbole, die Teil der VBA-Sprache sind. Schlüsselwörter haben genau festgeleg-

te spezielle Bedeutungen in VBA, und man kann sie nicht als Namen von Elementen wie zum Beispiel Prozeduren benutzen.

In diesem Fall wählt VBA den Namen der Prozedur – **OKButton_Click** – für uns aus. Der Name ist eine Kombination des Namens der Schaltfläche und des Ereignistyps.

Die letzte Zeile des automatisch erzeugten Codes lautet:

```
End Sub
```

Alle **Sub**-Prozeduren müssen mit dieser Zeile enden. Sie ist das Zeichen dafür, dass die Prozedur beendet werden soll.

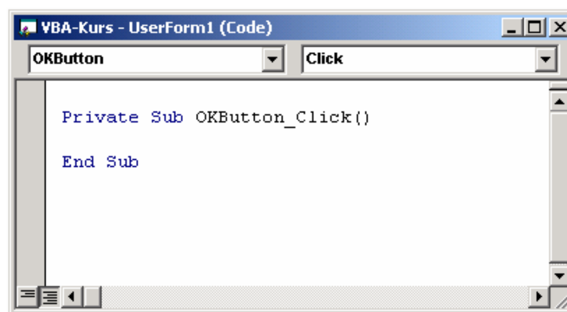


Abbildung 8 - Die Grundstruktur

Eigenen Code hinzufügen

Die Einfügemarke (Cursor) sollte zwischen zwei von VBA erstellten Codezeilen blinken. Indem wir gerade mal eine eigene Zeile einfügen, können wir die Schaltfläche so programmieren, dass sie das Programm beendet. Hier ist die Zeile:

```
Unload Me
```

Die Aussage **Unload** entfernt Formulare aus dem Speicher. Wenn man sich auf das Formular bezieht, an dessen Code man gerade arbeitet, muss man den Namen des Formulars nicht näher bezeichnen. Man kann es einfach **Me** nennen. Das ist ein spezieller VBA-Fachausdruck, der sich auf das aktuelle Formular und dessen gesamten Code bezieht.

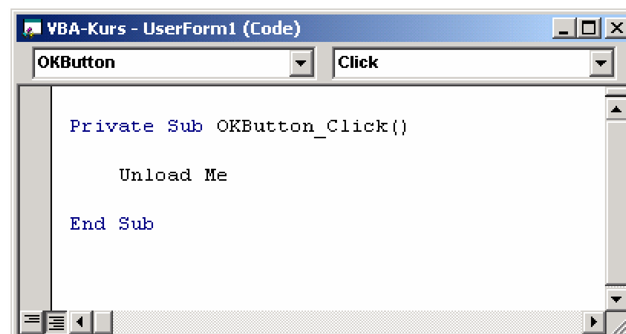


Abbildung 9 - Die fertige Prozedur

Eine zweite Prozedur erstellen

Die nächste Prozedur ist diejenige, die unsere Nachricht als Überschrift des Bezeichnungsfeldes anzeigt. Sie ist nur ein wenig komplizierter. Diese Prozedur soll aktiv werden, wenn das Formular auf dem Bildschirm erscheint. In dem immer noch geöffneten Codefenster gehen wir wie folgt vor:

1. Klicke im oberen Bereich des Codefensters auf den Pfeil neben der Textbox links, in der immer noch **OKButton** stehen sollte. Wir sehen nun eine Liste von Elementen, die zu diesem Formular gehören:

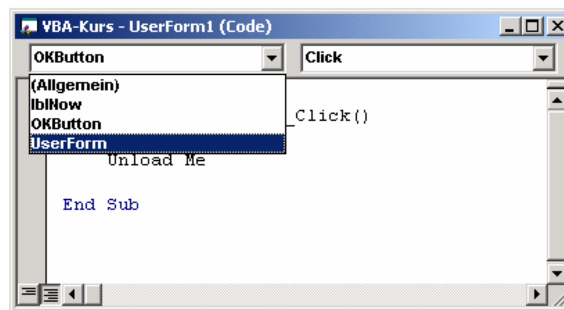


Abbildung 10 - Elemente in der Objektliste

2. Wähle das Element **UserForm** aus der **DropDown-Liste** aus. VBA erstellt eine neue Prozedur für das Ereignis „Klicken“. Das bedeutet gleichzeitig, dass jeder Nutzer, der auf irgendeinen beliebigen Teil des Formulars klickt, das kein Steuerelement hat, diese Prozedur auslöst. Im Beispielprogramm brauchen wir diese Prozedur nicht, daher müssen wir uns jetzt nicht damit beschäftigen.
3. Klicke auf den Pfeil rechts oben im Codefenster, um die DropDown-Liste mit den Prozeduren zu sehen. Diese Liste enthält alle Ereignisse, die VBA im Zusammenhang mit der UserForm entdecken kann. Sie ist recht lang. Hier bekommt man einen Eindruck davon, war man alles mit dem Programm machen kann.

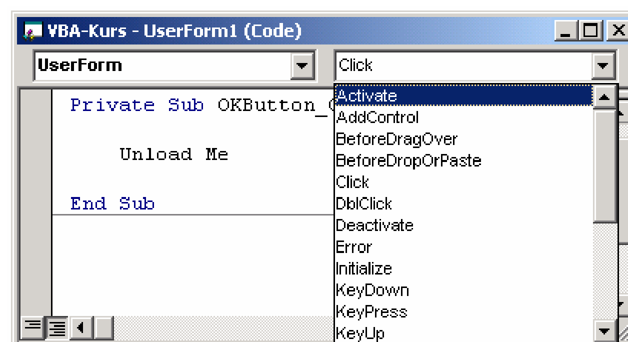


Abbildung 11 - Elemente in der Ereignisliste

4. Wähle das erste Element der Liste aus, das mit **Activate** bezeichnet ist. Man muss in der Liste nach oben blättern, um es zu finden. VBA erstellt pflichtbewusst das Grundgerüst der **UserForm_Activate-Prozedur**, die ausgelöst wird, wenn das Formular in den Speicher geladen wird.
5. Nun können wir die **UserForm_Click-Prozedur** löschen, indem wir den ganzen Text der Prozedur markieren und die Löschtaste betätigen. Dieser Schritt ist nicht zwingend notwendig, denn der Code stört nicht.

Den Code für die neue Prozedur schreiben

Da wir das Grundgerüst für die korrekte Ereignisprozedur schon haben, können wir nun daran gehen, es mit weiteren notwendigen Elementen auszufüllen. Gib drei neue Codezeilen zwischen den bereit vorhandenen ein. Die Prozedur sollte am Ende so aussehen:

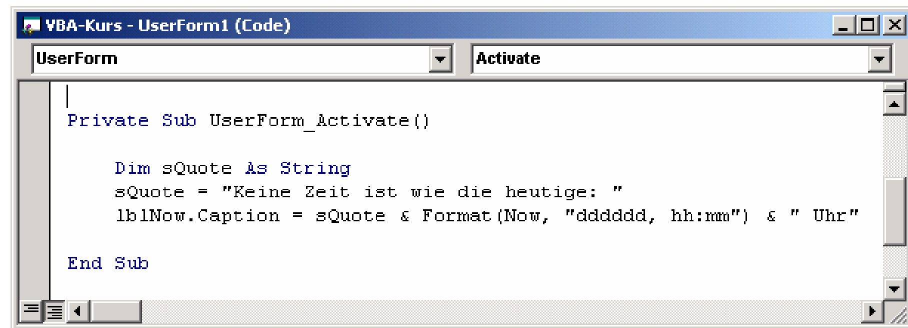


Abbildung 12 - Die neuen Codezeilen

Und nun wird erklärt, was das alles bedeutet:

1. Die erste Zeile, die wir eingeben, erstellt eine Variable namens **sQuote** und definiert sie als **String**. Ein String ist eine fortlaufende Abfolge von Textzeichen.
2. Die nächste Zeile, **sQuote = "Keine Zeit ist wie die heutige: "**, speichert den Text **Keine Zeit ist wie die heutige:** in der gerade erstellten **sQuote**-Variablen. Genauso wie in der Mathematik wird das Gleichheitszeichen hier dazu benutzt, um einer Variablen einen bestimmten Text zuzuordnen. Beachte auch das Leerzeichen vor dem abschließenden Anführungszeichen, das den einen Teil des Textes vom nachfolgenden Text trennt.
3. Die letzte Zeile, **lblNow.Caption = sQuote & Format(Now, "dddddd, hh:mm") & " Uhr"**, enthält den Code, der notwendig ist, um die vollständige Nachricht im Formular anzuzeigen.
4. Am Anfang der Zeile wird das Bezeichnungsfeld mit dem Namen **lblNow** als das Objekt, um das es hier geht, gekennzeichnet. Als Nächstes kommt der Punkt, der zeigt, dass die nun folgende Angabe eine zu **lblNow** gehörende Eigenschaft ist, in diesem Fall ist es die Überschrift. Eine Eigenschaft ist wie eine Variable, denn man kann ihren Wert ändern, während das Programm läuft. Daher benutzt man das Gleichheitszeichen, um die Eigenschaftswerte im Code zu ändern.
5. Der Rest der Zeile definiert die Nachricht, die als Überschrift des Bezeichnungsfeldes erscheint.
6. Der erste Teil der Nachricht ist die Variable **sQuote**. Das darauf folgende Zeichen (kaufmännisches Und &) weist VBA an, dem in sQuote gespeicherten Text das hinzuzufügen, was noch folgt.
7. Die **Now**-Funktion innerhalb der Klammern sorgt dafür, dass VBA sich die aktuelle Zeit (inkl. Datum) von der Uhr des Computers zu besorgen.
8. Die **Format**-Funktion nimmt diese Rohinformationen und wandelt sie in ein lesbares Format um. Die Buchstaben innerhalb der Anführungszeichen legen das spezielle Format fest, das wir im Fenster sehen. Dies ist eine mächtige VBA-Funktion, deren gesamtes Leistungsspektrum in der Online Hilfe nachzulesen ist.

Starte das Programm mit F5 und betrachte das Ergebnis!

Kapitel 3

EIN VBA-PROGRAMM ÄNDERN

Das bisher erstellte Programm ist eher einfach gestrickt und auch nicht unbedingt sehr funktionell. Das wollen wir ändern. In der „erweiterten“ Version soll das Programm folgenden Anforderungen gerecht werden:

1. Eingabe von Adressendaten
2. Eingabe eines Geburtsdatums
3. Überprüfen von Eingaben und –längen.
4. Errechnen und Anzeigen des Alters.

Die OK-Schaltfläche bleibt bestehen, nur beendet sie jetzt nicht (direkt) das Programm, sondern startet die Überprüfung der Eingaben, errechnet das Alter und zeigt dieses an. Erst danach wird das Programm beendet.

Das Bezeichnungsfeld **lblNow** kann gelöscht werden, da wir es nicht mehr benötigen. Gleichzeitig muss auch der zugehörige Code (in der Prozedur UserForm_Activate) entfernt werden, da es sonst zu einem Programmfehler kommt. Wenn nämlich das Bezeichnungsfeld von dem Formular entfernt wurde, führt die Codezeile

lblNow.Caption = sQuote & Format(Now, "dddddd, hh:mm") & " Uhr" unweigerlich zu einem Fehler, da hier auf ein Objekt **lblNow** mit dessen Eigenschaft **Caption** verweist, die es nicht mehr gibt!

Anpassen des Formulars

Um die Adressdaten eingeben lassen zu können, benötigen wir ein neues Steuerelement, das **Textfeld**. Und damit der Nutzer auch weiß, was jeweils einzugeben ist, muss vor jedes Textfeld ein Bezeichnungsfeld, in dessen Überschrift jeweils steht, was in das Textfeld einzugeben ist.

Das Formular muss hierfür etwas vergrößert werden, um alle Steuerelemente aufnehmen zu können.

Insgesamt sind Textfelder für folgende Daten einzufügen:

- Nachname
- Vorname

- Straße und Hausnummer
- Postleitzahl und Ort
- Geburtsdatum

Das angepasste Formular sollte in etwa wie folgt aussehen:

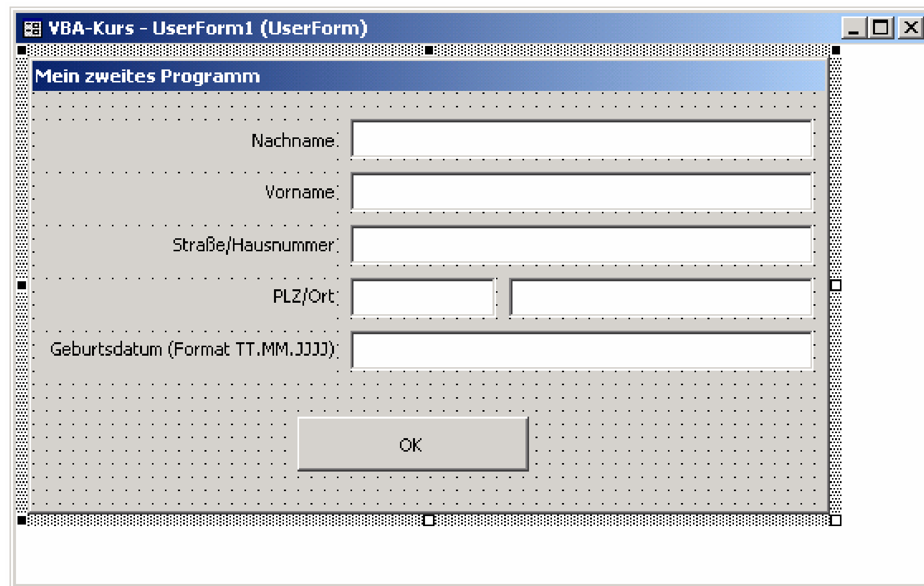


Abbildung 13 - Das angepasste Formular

Die Steuerelemente sind wie folgt zu bezeichnen

Bezeichnungsfelder	Textfelder
lblNachname	txtNachname
lblVorname	txtVorname
lblStrasse	txtStrasse
lblPlzOrt	txtPlz und txtOrt
lblGebDatum	txtGebDatum

Die OK-Schaltfläche wird umbenannt in **cmdOK**.

Textfelder sind in ihrem „Fassungsvermögen" theoretisch unbeschränkt, daher ist es notwendig, Maximallängen für die Eingabe zu definieren. Diese Werte werden im Eigenschaftenfenster einer jeder Textbox zur Eigenschaft MaxLength eingegeben. Standardmäßig ist hier immer der Wert 0 angegeben, was soviel bedeutet wie unbeschränkt.

Wir definieren also folgende Maximallängen für die Eingabe in den Textboxen:

Textfelder	Maximallänge
txtNachname	50
txtVorname	40
txtStrasse	50

Textfelder	Maximallänge
txtPlz und txtOrt	5 bzw. 40
txtGebDatum	10

Somit brauchen wir uns bei der Plausibilisierung der Eingabewerte nicht mehr um mögliche Längenüberschreitungen zu kümmern; dies übernimmt VBA für uns.

Was VBA jedoch nicht für uns übernimmt, ist die Prüfung der Befüllung von Pflichtfeldern oder Minimallängen.

In unserem zweiten Beispiel setzen wir fest, dass jedes Feld gefüllt sein muss. Darüber hinaus muss das Postleitzahlenfeld genau 5 Zeichen beinhalten, von denen alle numerisch sind.

Als letztes muss das Textfeld für das Geburtsdatum exakt 10 Zeichen lang sein und ein gültiges Datum beinhalten (TT.MM.JJJJ).

Die Plausibilitätsprüfungen können an verschiedenen Stellen durchgeführt werden:

- Bei jedem Tastendruck
- Beim „Verlassen" eines Steuerelements
- Zentral über Betätigung der OK-Schaltfläche

Wann eine Plausibilitätsprüfung stattfinden soll, ist zum einen Geschmacksache und zum anderen abhängig von der Art der Plausibilitätsprüfung. Grundsätzlich sollte man eigentlich so „schnell wie möglich" reagieren, wenn eine Eingabe falsch oder unvollständig ist und nicht erst warten, bis alle anderen Eingaben durchgeführt wurden um dann eine Reihe von Fehlermeldungen abspulen zu lassen. Das bleibt jedem selbst überlassen.

Besser als eine Fehlermeldung ist die Vermeidung von Fehlern. In unserem Programm ist die rein numerische Eingabe der Postleitzahl ein Paradebeispiel für Fehlervermeidung. Dies erreichen wir dadurch, dass wir ausschließlich nur numerische Eingaben erlauben.

Plausibilisierung der Postleitzahl

Um dies zu erreichen, müssen wir auf jeden Tastendruck reagieren. Das zugehörige Ereignis heißt **KeyPress**. Im Code-Editor müssen wir also in der linken Liste das Objekt **txtPlz** und in der rechten Liste das Ereignis **KeyPress** auswählen; VBA erzeugt sofort den notwendigen Prozedurkorpus:

```
Private Sub txtPlz_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)
End Sub
```

Neu sind hier die Angaben in der Klammer hinter der Ereignisbezeichnung **KeyPress**:

```
ByVal KeyAscii As MSForms.ReturnInteger
```

Interessant ist hier zunächst nur die Variable **KeyAscii**, die nichts anderes zu tun hat, als der Ereignisprozedur den sog. ASCII-Wert der gerade gedrückten Taste zu übergeben.

Jedes Zeichen auf der Tastatur lässt sich numerisch eindeutig darstellen; man spricht auch von dem **Ascii-Code** des Zeichens. Es mag etwas verwirrend sein, aber auch die numerischen Werte auf der Tastatur besitzen einen eigenen Ascii-Code. Wird also die Taste 5 gedrückt, übergibt die Variable KeyAscii nicht den Wert 5 sondern den Ascii-Code 53!

Hier eine kleine Auswahl:

Zeichen	ASCII-Code
0	48
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57
:	58
;	59
<	60
=	61
>	62
?	63
@	64
A	65
B	66
C	67
D	68
E	69
F	70
G	71
H	72
I	73
J	74
K	75
L	76
M	77
N	78
O	79

Zeichen	ASCII-Code
P	80
Q	81
R	82
S	83
T	84
U	85
V	86
W	87
X	88
Y	89
Z	90
[	91
\	92
]	93
^	94
_	95
`	96
a	97
b	98
c	99
d	100
e	101
f	102
g	103
h	104
i	105
j	106
k	107
l	108
m	109
n	110
o	111

Zeichen	ASCII-Code
p	112
q	113
r	114
s	115
t	116
u	117
v	118
w	119
x	120
y	121
z	122

Abbildung 14 - ASCII-Tabelle

Von diesen Codes interessieren uns für unsere Prüfung zunächst nur die Codes für die die Wert 0 bis 9 (= 48 bis 57). Also müssen wir überprüfen, ob entsprechende Tasten gedrückt worden sind.

Dies erfolgt über die sog. Select Case – Anweisung; also ungefähr so:

```
Select Case KeyAscii
   Case 48
   Case 49
   Case 50
   Case 51
   Case 52
   Case 53
   Case 54
   Case 55
   Case 56
   Case 57
End Select
```

Jetzt können wir jede einzelne numerische Eingabe von 0 bis 9 abfangen; nur hilft uns das eigentlich nicht, denn diese Eingaben sind ja OK. Hier benötigen wir also einen Fall (Case) für „den Rest". Dies wird als **Case Else** dargestellt. Außerdem ist es unschön, 10 Zeilen schreiben zu müssen, die alle „nichts bewirken"; aber auch hier gibt es Abhilfe:

```
Select Case KeyAscii
   Case 48 To 57
   Case Else
End Select
```

Jetzt wissen wir wann eine Zahl zwischen 0 und 9 oder eine andere Taste gedrückt wurde und können entsprechend reagieren. Zum einen können wir bei jeder Falscheingabe eine Nachricht anzeigen, die auf die Falscheingabe hinweist, nur verhindern wir sie dadurch nicht.

Deutlich eleganter ist es, nur Werte zwischen 0 und 9 zu akzeptieren und alle anderen Werte zu verwerfen. Dies erfolgt durch einen Trick: die Variable **KeyAscii** liefert nicht nur den aktuellen Zeichen-Code, sondern nimmt auch veränderte Werte entgegen und setzt diese um.

Theoretisch könnten wir also jede Falscheingabe durch die Rückgabe in einen anderen Wert verändern, nur macht das hier keinen Sinn. Das Steuerelement soll lediglich Zahlen zwischen 0 und 9 zulassen, alle anderen Zeichen müssen verworfen werden; dies erfolgt durch den Ascii-Code 0!

Die fertige Routine sieht dann so aus:

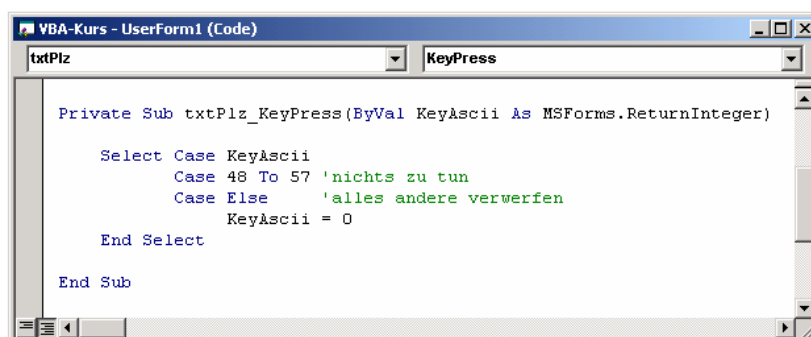


Abbildung 15 - Die fertige Routine

Es kann also bei diesem Steuerelement niemals zur Falscheingabe kommen!

Plausibilisierung der Texteingaben

Die Felder Nachname, Vorname, Straße und Ort können nicht auf die gleiche Art wie die Postleitzahl plausibilisiert werden. Hier ist es lediglich notwendig, die Pflichteingabe zu prüfen, was nichts anderes bedeutet, als die Länge der Eingabe auf > 0 zu überprüfen.

Da auch Leerzeichen als „vollwertige Zeichen“ gelten, ist es ratsam, vor der Längenüberprüfung alle führenden und abschließenden Leerzeichen zu entfernen, um wirklich nur noch die Textlängen überprüfen zu können. Dies lässt sich quasi „automatisieren“, in dem nach jeder Eingabe in einer dieser Felder mögliche Leerzeichen entfernt werden.

Dazu müssen wir wissen, wann eine Eingabe beendet ist; dies ist in der Regel dann der Fall, wenn ein Textfeld „verlassen“ wird. VBA stellt uns für dieses Ereignis die Routine **AfterUpdate** zur Verfügung. Diese Ereignisprozedur wird immer dann aufgerufen, wenn ein Eingabefeld verlassen wird und sich zuvor die Inhalte der Textbox verändert haben.

Wir benötigen also für die vier Textfelder Nachname, Vorname, Straße und Ort jeweils die Ereignisprozeduren **AfterUpdate**:

```
Private Sub txtNachname_AfterUpdate()

End Sub
Private Sub txtVorname_AfterUpdate()

End Sub
Private Sub txtStrasse_AfterUpdate()

End Sub
Private Sub txtOrt_AfterUpdate()

End Sub
```

Jetzt könnten wir im jeden Feld überprüfen, ob sich am Anfang oder am Ende der Eingabe ein oder mehrere Leerzeichen befinden. Das wäre aber sehr aufwendig und unnötig zeitintensiv. Aus diesem Grund benutzen wir eine VBA-Funktion die dies für uns übernimmt und auch gleichzeitig alle Leerzeichen am Anfang und am Ende eines **Strings** entfernt; es handelt sich um die **Trim$-Funktion**.

Das sog. „Trimmen“ von Strings wird sehr oft benötigt, da derartige Prüfungen immer wieder vorkommen.

In unserem Fall „trimmen“ wir einfach die aktuellen Werte der Steuerelemente – unabhängig davon, ob sich Leerzeichen im jeweiligen Steuerelement befinden oder nicht. Grundsätzlich erfolgt dies nach folgendem Schema:

Wert des Steuerelemente = Trimmen(Wert des Steuerelements) bzw. etwas weniger abstrakt txtNachname.Value = Trim$(txtNachname.Value). Dies ist für alle vier Steuerelemente einzutragen, so dass wir folgende Ereignis-Routinen zur Verfügung haben:

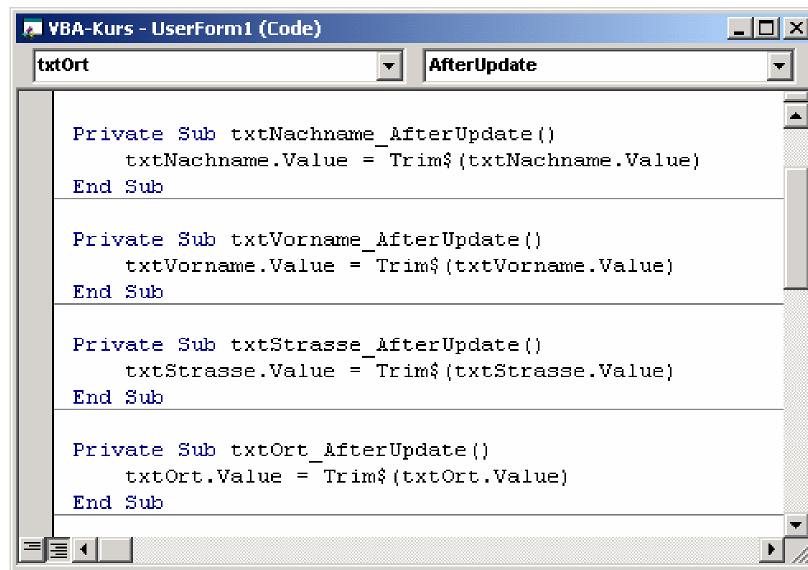


Abbildung 16 - Die veränderten Ereignis-Routinen

Auf diese Weise stellen wir sicher, dass die eingetragenen Daten „sauber" sind; dies ist u.a. dann relevant, wenn die Daten in Datenbanken abgespeichert werden.

Plausibilisierung des Geburtsdatums

Datumseingaben zu plausibilisieren ist sehr komplex und aufwendig. Theoretisch müsste man die Eingabe zunächst in die Werte für Tag, Monat und Jahr aufsplitten und dann sowohl technisch als auch logisch prüfen.

Eine technische Prüfung würde das Eingabeformat plausibilisieren, also:

Stellen 1+2	= numerisch
Stelle 3	= ein Punkt
Stellen 4+5	= numerisch
Stelle 6	= ein Punkt
Stellen 7-10	= numerisch

Diese technische Prüfung würde allerdings auch bei der Eingabe 99.99.9999 keinen Fehler finden, so dass eine fachliche Prüfung stattfinden muss. Diese ist recht komplex, denn hier müssen u.a. die Maximaltage je Monat unter Berücksichtigung von Schaltjahren etc. geprüft werden. Wir beschränken uns auf eine deutlich einfachere Methode der Datumsprüfung.

Für die Prüfung eines gültigen Datumswerts benutzen wir die VBA-Funktion **IsDate**().

Auszug aus der Online-Hilfe zur Funktion IsDate:

„IsDate gibt den Wert True zurück, wenn der Ausdruck ein Datum ist oder in ein gültiges Datum umgewandelt werden kann. Andernfalls wird False zurückgegeben. In Microsoft Windows liegen gültige Datumswerte im Bereich vom 1. Januar 100 n.Chr bis 31. Dezember 9999 n.Chr. Auf anderen Betriebssystemen können andere Bereiche gelten."

Diese Prüfung der Datumseingabe setzt jedoch voraus, dass der eingegebene Wert auch exakt 10 Zeichen lang ist (bzw. unserem Eingabeschema entspricht: TT.MM.JJJJ).

Um dies herauszufinden, bedient man sich der VBA-Funktion **Len**(). Mit dieser Funktion ermittelt man die Länge eines Strings.

Len-Funktion

Siehe auch Beispiel Zusatzinfo

Gibt einen Wert vom Typ **Long** zurück, der die Anzahl der Zeichen in einer Zeichenfolge oder die zum Speichern einer Variablen erforderlichen Bytes enthält.

Syntax

Len(*Zeichenfolge* | *Variablenname*)

Die Syntax der **Len**-Funktion besteht aus folgenden Teilen:

Teil	Beschreibung
Zeichenfolge	Beliebiger gültiger Zeichenfolgenausdruck. Wenn *Zeichenfolge* den Wert **Null** enthält, wird Null zurückgegeben.
Variablenname	Beliebiger gültiger Name für eine Variable. Wenn *Variablenname* den Wert **Null** enthält, wird Null zurückgegeben. Wenn *Variablenname* ein Wert vom Typ **Variant** ist, wird er von **Len** genauso behandelt wie ein Wert vom Typ **String**, und es wird immer die Anzahl der darin enthaltenen Zeichen zurückgegeben.

Bemerkungen

Eins (und nur eins) der zwei möglichen Argumente muß angegeben werden. Mit benutzerdefinierten Typen gibt **Len** die Größe so zurück, wie sie in die Datei geschrieben wird.

Anmerkung Die **LenB**-Funktion wird für die Byte-Daten verwendet, die in einer Zeichenfolge enthalten sind. **LenB** gibt nicht die Anzahl der Zeichen in einer Zeichenfolge zurück, sondern die Anzahl der Bytes, die zur Darstellung dieser Zeichenfolge verwendet werden. Mit benutzerdefinierten Typen gibt **LenB** die Größe im Speicher zurück, einschließlich aller Polster zwischen Elementen.

Anmerkung Die **Len**-Funktion gibt nicht immer die tatsächlich zum Speichern benötigten Bytes zurück, wenn sie mit Zeichenfolgen variabler Länge in benutzerdefinierten Datentypen verwendet wird.

Abbildung 17 - Auszug aus der OL-Hilfe: LEN-Funktion

Nun sind wir in der Lage, mithilfe der VBA-Funktionen **Len()** und **IsDate()** die Datumsplausibilisierung durchzuführen. Wenn wir jetzt feststellen, dass die Eingabe **nicht** korrekt ist, sollten wir den Nutzer auch darauf hinweisen. Dies erfolgt in der Regel durch Hinweise, die in einem separaten Fenster angezeigt werden.

Auch hier hilft uns VBA weiter und stellt uns die Funktion **MsgBox** zur Verfügung.

MsgBox-Funktion

Siehe auch Beispiel Zusatzinfo

Zeigt eine Meldung in einem Dialogfeld an und wartet darauf, daß der Benutzer auf eine Schaltfläche klickt. Es wird dann einen Wert vom Typ **Integer** zurückgegeben, der anzeigt, auf welche Schaltfläche der Benutzer geklickt hat.

Syntax

MsgBox(*prompt*[, *buttons*] [, *title*] [, *helpfile*, *context*])

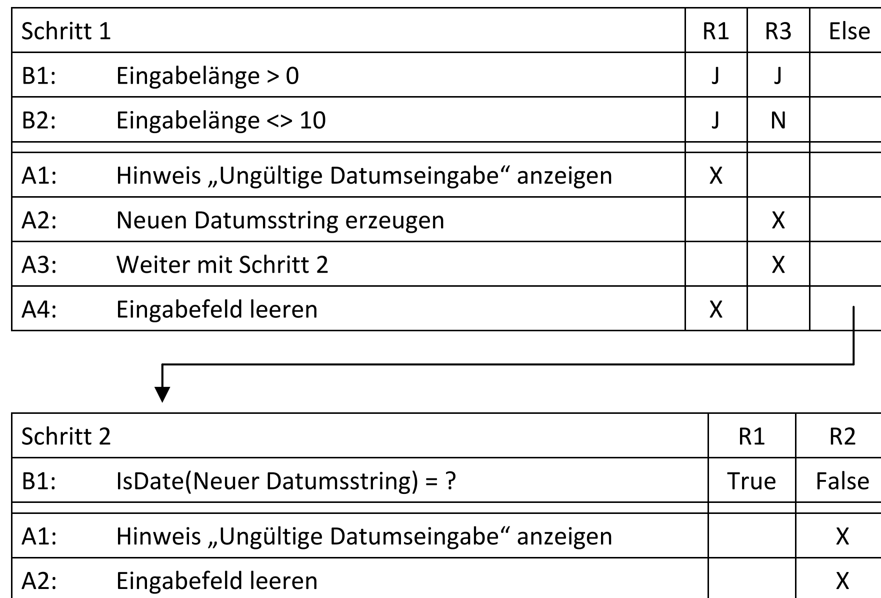
Die Syntax der **MsgBox**-Funktion verwendet die folgenden benannten Argumente:

Teil	Beschreibung
prompt	Erforderlich. Ein Zeichenfolgenausdruck, der als Meldung im Dialogfeld erscheint. Die Maximallänge von ***prompt*** ist - je nach Breite der verwendeten Zeichen - etwa 1024 Zeichen. Wenn ***prompt*** aus mehreren Zeilen besteht, müssen Sie die Zeilen mit einem Wagenrücklaufzeichen (**Chr(**13**)**), einem Zeilenvorschubzeichen (**Chr(**10**)**) oder einer Kombination aus Wagenrücklaufzeichen und Zeilenvorschubzeichen (**Chr(**13**)** & **Chr(**10**)**) trennen.
buttons	Optional. Ein numerischer Ausdruck, der der Summe der Werte entspricht, die Anzahl und Typ der anzuzeigenden Schaltflächen, die Art des zu verwendenden Symbols sowie die Standardschaltfläche und die Bindung des Dialogfeldes angeben. Wenn Sie ***buttons*** nicht angeben, ist der Standardwert 0.
title	Optional. Ein Zeichenfolgenausdruck, der in der Titelleiste des Dialogfeldes angezeigt wird. Wenn Sie ***title*** nicht angeben, wird der Anwendungsname in der Titelleiste angezeigt.
helpfile	Ein Zeichenfolgenausdruck, der die Hilfedatei mit der kontextbezogenen Hilfe für das Dialogfeld angibt. Wenn Sie ***helpfile*** angeben, müssen Sie auch ***context*** angeben.
context	Optional. Ein numerischer Ausdruck mit der Hilfekontextkennung, die der Autor der Hilfe für das entsprechende Hilfethema gegeben hat. Wenn Sie ***context*** angeben, müssen Sie auch ***helpfile*** angeben.

Abbildung 18 - Auszug aus der OL-Hilfe: MsgBox-Funktion

Hinweis: Wenn bei solchen Hilfe-Seiten Argumente in eckigen Klammern stehen, so handelt es sich um optionale Argumente, d.h. diese müssen nicht zwingend benutzt werden. Im Fall der MsgBox-Funktion ist also lediglich der Hinweistext (prompt) notwendig.

Die Plausibilität der Datumsprüfung entspricht nun folgender Logik:



Schritt 1		R1	R3	Else
B1:	Eingabelänge > 0	J	J	
B2:	Eingabelänge <> 10	J	N	
A1:	Hinweis „Ungültige Datumseingabe“ anzeigen	X		
A2:	Neuen Datumsstring erzeugen		X	
A3:	Weiter mit Schritt 2		X	
A4:	Eingabefeld leeren	X		

Schritt 2		R1	R2
B1:	IsDate(Neuer Datumsstring) = ?	True	False
A1:	Hinweis „Ungültige Datumseingabe“ anzeigen		X
A2:	Eingabefeld leeren		X

oder als Code:

```
Private Sub txtGebDatum_AfterUpdate()

    Dim sDate As String

   'Eingabe "trimmen"
    sDate = Trim$(txtGebDatum.Value)
   'Länge 0 prüfen
    If Len(sDate) > 0 Then
       'Länge <> 10 prüfen
        If Len(sDate) <> 10 Then
           'Hinweis anzeigen
            MsgBox "Ungültige Datumseingabe. Bitte das Format TT.MM.JJJJ verwenden!"
           'Eingabefeld leeren
            txtGebDatum.Value = ""
        Else
           'Gültigkeit des Datums überprüfen
            If Not IsDate(sDate) Then
               'Hinweis anzeigen
                MsgBox "Ungültige Datumseingabe. Bitte das Format TT.MM.JJJJ verwenden!"
               'Eingabefeld leeren
                txtGebDatum.Value = ""
            End If
        End If
    End If

End Sub
```

Abbildung 19 - Die Plausibilität der Datumsprüfung

Wie in diesem Beispiel sollte man grundsätzlich seinen Code gut dokumentieren („grüner“ Beschreibungstext). Diese sog. Kommentarzeilen werden durch ein Hochkomma eingeleitet und haben keinen Einfluss auf den Programmverlauf.

Neu ist hier die Bedingungsprüfung **If – Then – Else**. Diese WENN – DANN – SONST Konstruktion ist relativ selbsterklärend und für die einfache Nutzung wie in diesem Beispiel ist lediglich darauf zu achten, dass am Ende jeder **If-Bedingung** ein **Then** steht und das jeder Bedingungsblock mit einen **End If** abgeschlossen werden muss.

Alter berechnen

Nachdem wir nun die Eingaben plausibilisiert haben, können wir anhand des Geburtsdatums (sofern vorhanden) das Alter errechnen.

Im einfachsten Fall müssen wir lediglich das Jahr ermitteln und vom aktuellen Jahr abziehen. Korrekt ist es jedoch, auch Monate und Tage zu berücksichtigen, um auch das aktuell korrekte Alter zu ermitteln.

Lösung für „den Hausgebrauch“ (simple Lösung):

In diesem Fall wird lediglich das Jahr als Basis des Alters benutzt, auch wenn das bedeutet, das Alter falsch zu berechnen, da der Geburtstag noch nicht erreicht ist. Bevor wir hier nun wieder mit Stringfunktionen versuchen, das Jahr zu extrahieren (das wäre: **Mid$(sDate, 7, 4)**), benutzen wir hierzu zwei weitere VBA-Funktionen, die uns diese Arbeit abnehmen.

Zunächst müssen wir aus dem String mit dem Geburtsdatum einen „echten“ Datumswert erzeugen.

Dazu stellt uns VBA die Konvertierungsfunktion **CDate**() zur Verfügung. Dieser Funktion wird als Argument der uns bekannte Datumsstring übergeben und wir erhalten den konvertierten String vom Typ Date zurück.

Beispiel:

```
Dim sDatum As String
Dim dtDatum As Date

sDatum = "01.01.1970"
dtDatum = CDate(sDatum)
```

Nachdem wir das Geburtsdatum als „echtes“ Datum zur Verfügung haben, können wir mit einer der verfügbaren Datumsfunktionen von VBA das Jahr ermitteln. In diesem Fall benötigen wir die **Year**-Funktion. Dieser Funktion wird ein Datum übergeben und man erhält als Rückgabewert das Jahr.

```
    Jahr = Year(dtDatum)
```

Nachdem wir das Jahr des Geburtstags ermittelt haben, benötigen wir noch das aktuelle Jahr für den Vergleich. Auch hierzu können wir die **Year**()-Funktion benutzen, nur müssen wir jetzt das Tagesdatum übergeben.

VBA stellt uns hier die Funktion **Date**() zur Verfügung, die das aktuelle Datum übergibt. Für unser Beispiel bedeutet das:

```
Dim dtDate As Date
dtDate = CDate(txtGebDatum.Value)
MsgBox "Sie sind/werden " & Year(Date) - Year(dtDate) & " Jahre alt!"
```

In dieser Hinweisanzeige werden verschiedene Informationen zu einem Text zusammengefasst (=konkateniert), in dem die einzelnen Hinweis“teile“ mit dem &-Zeichen zusammengefügt werden!

Unsere Routine hat sich wie folgt verändert:

```
Private Sub txtGebDatum_AfterUpdate()

    Dim sDate   As String
    Dim dtDate  As Date

   'Eingabe "trimmen"
    sDate = Trim$(txtGebDatum.Value)
   'Länge 0 prüfen
    If Len(sDate) > 0 Then
       'Länge <> 10 prüfen
        If Len(sDate) <> 10 Then
           'Hinweis anzeigen
            MsgBox "Ungültige Datumseingabe. Bitte das Format TT.MM.JJJJ verwenden!"
           'Eingabefeld leeren
            txtGebDatum.Value = ""
        Else
           'Gültigkeit des Datums überprüfen
            If Not IsDate(sDate) Then
               'Hinweis anzeigen
                MsgBox "Ungültige Datumseingabe. Bitte das Format TT.MM.JJJJ verwenden!"
               'Eingabefeld leeren
                txtGebDatum.Value = ""
            Else
                dtDate = CDate(txtGebDatum.Value)
                MsgBox "Sie sind/werden " & Year(Date) - Year(dtDate) & " Jahre alt!"
            End If
        End If
    End If

End Sub
```

Abbildung 20 - Die veränderte Routine

Hier wurde jetzt die Deklaration der Variablen **dtDate** vom Type **Date** hinzugefügt.

Die Deklaration von Variablen (**Dim**) erfolgt in der Regel zu Beginn einer jeden Routine.
Weiterhin wurden die Errechnung und die Anzeige des Alters in einem neuen Else-Zweig dort eingefügt, wo die Eingabe bereits entsprechend plausibilisiert wurde.

Der Nachteil dieser Lösung liegt in dem Umstand, dass jedes Mal diese Altersberechnung und –anzeige erfolgt, wenn diese Routine aufgerufen wird. Das ist nicht sehr elegant und sollte daher an anderer Stelle erfolgen. Aus diesem Grund werden wir die Berechnung des Alters in eine eigene Funktion auslagern.

Eine Funktion gibt – im Gegensatz zu einer sog. Sub-Routine – einen Wert zurück. In diesem Fall möchten wir der Funktion den Eingabestring übergeben und das errechnete Alter zurückbekommen.

Derartige Funktionen müssen wir selbst erstellen; daher schreiben wir an eine „freie" Stelle im Codefenster:

```
Private Function Alter(sDatum As String) As Long

End Function
```

Abbildung 21 - Der Grundaufbau unserer Prozedur Alter

Die Funktion Alter erhält als Argument einen String (mit der Datumseingabe) und gibt einen numerischen Wert (hier: Typ Long) zurück.

Da es sich um eine eigenständige Funktion handelt, sollten wir innerhalb der Funktion (selbst nochmals) den Stringwert plausibilisieren, da wir nie davon ausgehen dürfen, dass dies bereits geschehen ist und wir „blind" darauf vertrauen können.

Demzufolge stellt sich diese (vereinfachte) Altersberechnungsfunktion wie folgt dar:

```
Private Function Alter(sDatum As String) As Long

    Dim dtDate As Date
    sDatum = Trim$(sDatum)
    If Len(sDatum) = 10 Then
        If IsDate(sDatum) Then
            dtDate = CDate(sDatum)
            Alter = Year(Date) - Year(dtDate)
        End If
    End If

End Function
```

Abbildung 22 - Die "einfache" Altersberechnung

Nur wenn alle Plausibilitäten erfolgreich durchlaufen wurden, wird das Alter errechnet und zurückgegeben. In allen anderen Fällen erhält man als Rückgabewert 0, dem Standardwert für den Long-Datentyp.

Da wir jetzt eine eigene Funktion für die Altersberechnung haben, müssen wir den zuvor eingefügten Else-Zweig (s.o.) wieder entfernen.

Diese Lösung hat bekanntermaßen den Nachteil, das Alter nicht unbedingt korrekt zu errechen, da lediglich das Jahr und nicht die Monate und Tage berücksichtigt werden.

Es steht jetzt jedem frei, dies zu ändern und die Funktion entsprechend anzupassen, in dem die Tage und der Monat ermittelt werden, um diese Werte gegen das aktuelle Tagesdatum zu prüfen, um das korrekte Alter zu ermitteln...

Profilösung

Die nachfolgende Lösung ist für fortgeschrittene VBA-Programmierer gedacht und wird daher nicht weiter kommentiert:

```
Function Alter(dtGebDat As Date) As Long

    Alter = Year(Now) - Year(dtGebDat) + (DateSerial(Year(Now), Month(dtGebDat), Day(dtGebDat)) > Now)

End Function
```

Abbildung 23 - Die Profi-Altersberechnung

Zum Abschluss benötigen wie jetzt noch den Code für die OK-Schaltfläche.

Dort finden weitere Prüfungen statt und – wenn alles OK ist – wird das Alter angezeigt und nachfolgend das Programm beendet.

Zu Beginn wurde die Anforderung formuliert, dass alle Felder Pflichteingaben sind. Dies hätte man bei den „reinen“ Texteingaben direkt durchführen können, nur wollen wir dies zentral an einer Stelle durchführen.

Dabei prüfen wir (nochmals) die Feldinhalte und erzeugen einen entsprechenden Hinweis, der auf die Fehler aufmerksam macht. Hier wollen wir nun alle Fehler auf einmal anzeigen,

damit der Nutzer nicht mit mehreren Hinweisen konfrontiert wird, die im Grunde genommen die gleiche Aussage beinhalten.

Diese letzte Aufgabe beschäftigt sich hauptsächlich mit String-Operationen, da wir je nach Fehleranzahl eine unterschiedliche Hinweismeldung erstellen müssen.

Sind alle Daten OK, so wird das Alter errechnet, angezeigt und das Programm anschließend beendet.

Noch ein Tipp für die Hinweismeldung: obwohl man Strings in „beliebiger" Länge aneinanderreihen kann, sollte man doch hier und da einen „Zeilenumbruch" einfügen, um die „Lesbarkeit" zu erhöhen. Einen Zeilenumbruch wird durch die Kombination von zwei „Zeichen" erreicht:

```
Carriage Return + Line Feed (CR + LF)
```

Dies funktioniert in der gesamten Windows-Welt, nur „sieht" man diese Zeichen nicht, lediglich deren „Wirkung". Um diese „Wirkung" zu erreichen, müssen wir also diese beiden „Zeichen" zusammenführen und in unseren Hinweistext „einbauen".

Genauso wie alle anderen „sichtbaren" Zeichen besitzen auch diese nicht sichtbaren (Steuer)zeichen sog. Ascii-Codes: Carriage Return = 13 und Line Feed = 10.

Damit man nicht mit diesen kryptischen Zahlenwerten hantieren muss, stellt uns VBA eine Reihe von sog. Konstanten zur Verfügung, welche diese Werte beinhalten. Für unseren Zweck nutzen wir daher die VBA-Konstante **vbCrLf**:

```
Hinweis = Text1 & vbCrLf & Text2
```

Musterlösung

```
Option Explicit

Private Sub txtNachname_AfterUpdate()

    txtNachname.Value = Trim$(txtNachname.Value)

End Sub

Private Sub txtVorname_AfterUpdate()

    txtVorname.Value = Trim$(txtVorname.Value)

End Sub

Private Sub txtStrasse_AfterUpdate()

    txtStrasse.Value = Trim$(txtStrasse.Value)

End Sub

Private Sub txtOrt_AfterUpdate()

    txtOrt.Value = Trim$(txtOrt.Value)

End Sub

Private Sub txtPlz_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)

    Select Case KeyAscii
           Case 48 To 57 'nichts zu tun
           Case Else     'alles andere verwerfen
                KeyAscii = 0
    End Select

End Sub

Private Sub txtGebDatum_AfterUpdate()

    Dim sDate As String

   'Eingabe "trimmen"
    sDate = Trim$(txtGebDatum.Value)
   'Länge 0 prüfen
    If Len(sDate) > 0 Then
       'Länge <> 10 prüfen
        If Len(sDate) <> 10 Then
           'Hinweis anzeigen
            MsgBox "Ungültige Datumseingabe. Bitte das Format TT.MM.JJJJ verwenden!"
           'Eingabefeld leeren
            txtGebDatum.Value = ""
        Else
           'Gültigkeit des Datums überprüfen
            If Not IsDate(sDate) Then
               'Hinweis anzeigen
                MsgBox "Ungültige Datumseingabe. Bitte das Format TT.MM.JJJJ verwenden!"
               'Eingabefeld leeren
                txtGebDatum.Value = ""
            End If
        End If
    End If

End Sub
```

Abbildung 24 – Musterlösung: Die fertigen Routinen Teil 1

```
Private Sub cmdOK_Click()
    Dim sHinweis As String, lZaehler As Long, ctrControl As TextBox
   'Eingabe für das Feld [Nachname] überprüfen
    If Len(Trim$(txtNachname.Value)) = 0 Then
        If Len(sHinweis) > 0 Then sHinweis = sHinweis & ", " & vbCrLf
        sHinweis = sHinweis & "der Nachname"
        If ctrControl Is Nothing Then Set ctrControl = txtNachname
        lZaehler = lZaehler + 1
    End If
   'Eingabe für das Feld [Vorname] überprüfen
    If Len(Trim$(txtVorname.Value)) = 0 Then
        If Len(sHinweis) > 0 Then sHinweis = sHinweis & ", " & vbCrLf
        sHinweis = sHinweis & "der Vorname"
        If ctrControl Is Nothing Then Set ctrControl = txtVorname
        lZaehler = lZaehler + 1
    End If
   'Eingabe für das Feld [Strasse und Hausnummer] überprüfen
    If Len(Trim$(txtStrasse.Value)) = 0 Then
        If Len(sHinweis) > 0 Then sHinweis = sHinweis & ", " & vbCrLf
        sHinweis = sHinweis & "die Straße"
        If ctrControl Is Nothing Then Set ctrControl = txtStrasse
        lZaehler = lZaehler + 1
    End If
   'Eingabe für das Feld [Postleitzahl] überprüfen
    If Len(Trim$(txtPlz.Value)) = 0 Then
        If Len(sHinweis) > 0 Then sHinweis = sHinweis & ", " & vbCrLf
        sHinweis = sHinweis & "die Postleizahl"
        If ctrControl Is Nothing Then Set ctrControl = txtPlz
        lZaehler = lZaehler + 1
    End If
   'Eingabe für das Feld [Ort] überprüfen
    If Len(Trim$(txtOrt.Value)) = 0 Then
        If Len(sHinweis) > 0 Then sHinweis = sHinweis & ", " & vbCrLf
        sHinweis = sHinweis & "der Ort"
        If ctrControl Is Nothing Then Set ctrControl = txtOrt
        lZaehler = lZaehler + 1
    End If
   'Eingabe für das Feld [Geburtsdatum] überprüfen
    If Len(Trim$(txtGebDatum.Value)) = 0 Then
        If Len(sHinweis) > 0 Then sHinweis = sHinweis & ", " & vbCrLf
        sHinweis = sHinweis & "das Geburtsdatum"
        If ctrControl Is Nothing Then Set ctrControl = txtGebDatum
        lZaehler = lZaehler + 1
    End If
   '-----------------------------------------------------------------
   'Auswertung der vorherigen Prüfungen
   '-----------------------------------------------------------------
    If lZaehler = 1 Then
         sHinweis = _
            "Die Eingaben sind unvollständig, da eine Information fehlt:" & _
             vbCrLf & vbCrLf & sHinweis & "!"
         MsgBox sHinweis, vbExclamation, "Adressenverwaltung"
         ctrControl.SetFocus
    ElseIf lZaehler > 1 Then
         sHinweis = _
            "Die Eingaben sind unvollständig, da " & Str(lZaehler) & _
            " Informationen fehlen:" & vbCrLf & vbCrLf & sHinweis & "!"
         MsgBox sHinweis, vbExclamation, "Adressenverwaltung"
         ctrControl.SetFocus
    Else
        sHinweis = _
            "Die Eingaben sind OK!" & vbCrLf & vbCrLf & "Sie sind aktuell " & _
            Alter(CDate(txtGebDatum.Value)) & " Jahre alt!"
        MsgBox sHinweis, vbInformation, "Adressenverwaltung"
        Unload Me
    End If
End Sub
```

Abbildung 25 - Musterlösung: Die fertigen Routinen Teil 2

Der Dialog stellt sich wie folgt dar:

Abbildung 26 - Musterlösung: Der Dialog

In dieser Musterlösung wurde die „Profi-Variante für die Altersberechnung eingefügt. Hier können natürlich auch die Minimallösung oder selbst erstellte Funktionen benutzt werden.

Neu in dieser Musterlösung ist in der Prozedur **cmdOK_Click**() die Nutzung der **Objektvariablen ctrControl**.

Mit dieser Objektvariablen wird jeweils auf die Textbox referenziert, die leer ist. Damit hat man die Möglichkeit, nach allen Plausibilitätsprüfungen die erste Textbox automatisch zu aktivieren, die leer ist (Ergonomie). Dazu ist es notwendig, sich auch wirklich nur die erste leere Textbox zu merken. Wenn man sich alle leeren Textboxen merkt, so wird auch nur die letzte (leere) Textbox aktiviert. Man „merkt“ sich eine Textbox, in dem man über die Objektvariable auf die betroffene Textbox verweist:

```
Set ctrControl = txtNachname
```

Objekte werden grundsätzlich über die Set-Anweisung referenziert. Um auch wirklich nur die erste leere Textbox zu referenzieren, ist es notwendig, vor der Referenzierung zu prüfen, ob die Objektvariable bereits einen Verweis auf eine andere Textbox besitzt. In diesem Fall erfolgt keine erneute Referenzierung, und die erste Zuweisung bleibt bestehen.

Um zu prüfen, ob eine Objektvariable bereits eine Referenz auf eine Control (eigentlich: auf ein Objekt) besitzt, muss man den „Wert“ gegen „Nothing“ vergleichen. Besitzt eine Objektvariable den Wert „Nothing“, wurde sie noch nicht referenziert.

```
If ctrControl Is Nothing Then Set ctrControl = txtNachname
```

Wurde die Objektvariable referenziert, so verweist diese auf eine der Textboxen; und zwar auf die erste, leere Textbox. Um diese Textbox nun zu aktivieren (d.h. den Cursor in das Steuerelement setzen), reicht es aus, die Textbox eigene Funktion SetFocus über die Objektvariable zu benutzen:

```
ctrControl.SetFocus
```

Kapitel

4

TIPPS & TRICKS – ROUTINEN

Subs und Functions oder die Frage: Wie finde ich die richtigen Argumente?

Was ist eine Prozedur? Die Online-Hilfe informiert wie folgt:

Prozedur

Eine benannte Folge von Anweisungen, die als Einheit ausgeführt werden. **Function**, **Property** und **Sub** sind zum Beispiel Prozedurtypen. Der Name einer Prozedur wird immer auf Modulebene definiert. Der gesamte ausführbare Code muß in einer Prozedur enthalten sein. Prozeduren können nicht in andere Prozeduren eingesetzt werden.

Abbildung 27 - OL-Hilfe zum Thema Prozedur

oder auch:

- Eine **Sub**-Prozedur ist eine Folge von Visual Basic-Anweisungen, die in den Anweisungen **Sub** und **End Sub** eingeschlossen sind und Aktionen ausführen, aber keinen Wert zurückgeben. Eine **Sub**-Prozedur kann Argumente, z.B. Konstanten, Variablen oder Ausdrücke verwenden, die über eine aufrufende Prozedur übergeben werden. Wenn eine **Sub**-Prozedur über keine Argumente verfügt, muss die **Sub**-Anweisung ein leeres Klammernpaar enthalten.
- Eine **Function**-Prozedur ist eine Folge von Visual Basic-Anweisungen, die durch die Anweisungen **Function** und **End Function** eingeschlossen sind. Eine **Function**-Prozedur ähnelt einer Sub-Prozedur, kann aber auch einen Wert zurückgeben. Eine **Function**-Prozedur kann Argumente, wie z.B. Konstanten, Variablen oder Ausdrücke verwenden, die über die aufgerufene Prozedur übergeben werden. Wenn eine **Function**-Prozedur über keine Argumente verfügt, muss deren **Function**-Anweisung ein leeres Klammernpaar enthalten. Eine **Funktion** gibt einen Wert zurück, indem ihrem Namen ein Wert in einer oder mehreren Anweisungen der Prozedur zugewiesen wird.
- Eine **Property**-Prozedur ist eine Folge von Visual Basic-Anweisungen, die es einem Programmierer ermöglicht, benutzerdefinierte Eigenschaften zu erstellen und zu bearbeiten. **Property**-Prozeduren können zur Erstellung schreibgeschützter Eigenschaften für Formulare, Standardmodule und Klassenmodule verwendet werden. **Property**-Prozeduren sollten anstelle von Public-Variablen in Code verwendet werden, der ausgeführt werden muss, wenn der Wert der Eigenschaft festgelegt wird. Im Gegensatz zu Public-Variablen können **Property**-Prozeduren im Objektkatalog beschreibende Texte zugeordnet sein.

Alles klar? Schön. Dann können wir ja gleich in medias res gehen...

Sub-Routinen geben keinen Wert zurück, Function Routinen können dies jedoch. Richtig? Jain!

Das „Geheimnis“ liegt in der Art, wie Argumente übergeben werden.

Grundsätzlich unterscheidet man bei Argumenten zwischen der Übergabe einer **Referenz** und der Übergabe eines **Wertes**. Der Standard ist die Übergabe einer **Referenz**!

In der Deklaration von Argumenten kann dies explizit durch die Klassifizierung durch **ByVal** bzw. **ByRef** erfolgen.

```
Sub Foo(sMyValue As String)
ist exakt das gleiche wie
Sub Foo(ByRef sMyValue As String)
```

da ByRef („als Referenz“) der Standard in VB/VBA ist. Erst durch die Änderung in **ByVal** verändern wir das Argument auf entscheidende Art und Weise.

```
Sub Foo(ByVal sMyValue As String)
```

- ➔ Bei der Übergabe einer Referenz wird lediglich der „Zeiger“ auf eine Variable im Speicher übergeben.
- ➔ Bei der Übergabe als Kopie wird eine Kopie der Variable (nebst dessen Inhalt) übergeben.

Was wird in der Routine **Main** in der MsgBox angezeigt?

Hier wird (implizit) das Wissen vorausgesetzt, dass man andere **Subs** durch die **Call**-Anweisung aufrufen kann. Erstelle ein neues Modul im VBA-Editor, gib nebenstehenden Quellcode ein, führe **Sub Main**() aus und finde deine Annahme über die Meldung bestätigt (oder auch nicht...).

```
Sub Main()

    Dim sMyValue As String
    sMyValue = "Hallo Welt"

    Call Test01(sMyValue)
    Call Test02(sMyValue)

    MsgBox sMyValue

End Sub
---------------------------------------------
Sub Test01(ByRef sMyValue As String)

    sMyValue = "Das war's dann..."
    Call Test03(sMyValue)

End Sub
---------------------------------------------
Sub Test02(ByVal sMyValue As String)

    sMyValue = "Und was jetzt?"

End Sub
---------------------------------------------
Sub Test03(sMyValue As String)

    sMyValue = "Geben wir der Variablen den Rest..."

End Sub
```

Abbildung 28 - Code-Beispiel

Können Sub-Routinen jetzt Werte zurückgeben, oder können sie es nicht?

Die hier dargestellte ByRef- bzw. ByVal-Problematik gilt im Übrigen für sämtliche Prozedurtypen und ist nicht auf Sub-Routinen beschränkt!

Fazit: Man sollte grundsätzlich seine Argumente mit **ByVal** klassifizieren, um keine unliebsamen Überraschungen zu erfahren. Die Klassifizierung mit **ByRef** bzw. die Übernahme des Standards sollte nur bewusst in den Fällen erfolgen, bei denen man „etwas" damit bezweckt.

Was ist eine Property oder: Wie schütze ich meine Variablen?

Wir wissen, dass es neben Sub und Function Prozeduren sog. Property (Eigenschaften) Prozeduren gibt. Aber um welche bzw. wessen Eigenschaften geht es dabei überhaupt?

Im Rahmen dieser Einführung beschränken wir uns auf die einfache Nutzung von Property-Prozeduren zur „Kapselung" von Variablen.

```
Option Explicit

Dim sMeinDateiPfad As String

Sub Schreibe_in_die_Datei()
     Dim i As Integer
     Open sMeinDateiPfad For Input As #i
     '...
     Close #i
End Sub
```

In dieser Routine (**Schreibe_in_die_Datei**) wird direkt versucht, eine Datei zu öffnen, dessen Pfad in der Variable sMeinDateiPfad steht bzw. stehen sollte.

Ist dies jedoch nicht der Fall, kommt es hier zu einem Programmfehler. Jetzt kann man vor dem Öffnen der Datei zunächst prüfen, ob die Variable bereits den richtigen Dateipfad enthält. Das muss aber dann jedes Mal und in jeder Routine erfolgen, die über diese *modulglobale* Variable versuchen, auf die Datei zuzugreifen; dies ist äußerst umständlich und auch nicht ganz ungefährlich. Ändern sich nämlich der Dateipfad, muss dies in allen Routinen „nachgezogen" werden, die auf diesen Dateipfad zugreifen.

Die Lösung heißt: Kapselung der Variabelen-Initialisierung und –Prüfung innerhalb einer Property-Prozedur.

Hierbei werden allen notwendigen Schritte Property-Prozedur durchgeführt.

Die Routine **Schreibe_in_die_Datei** würde lediglich die Property-Prozedur aufrufen, die dann den korrekten Dateipfad zurückliefert. Es gibt grundsätzlich immer 2 Varianten einer Property-Prozedur: eine **Get** und eine **Let**-Variante.

- ➔ Die **Get** Variante funktioniert wie Funktion und gibt einen Wert zurück.
- ➔ Die **Let** Variante ist dafür zuständig, die betroffene Variable zu initialisieren.

Nennen wir unsere **Property** beispielsweise **MeinDateiPfad**. Damit wir nicht die Arbeit haben, jetzt direkt 2 Prozedurblöcke erstellen zu müssen, nutzen wir einen Assistenten, den VBA uns zur Verfügung stellt:

Über das Menü im **VBA-EDITOR EINFÜGEN|PROZEDUR**... rufen wir einen kleinen Dialog auf, in dem wir den Namen der Prozedur, seinen Typ und seinen Gültigkeitsbereich eintragen.

VBA erstellt uns nach Klick auf die OK-Schaltfläche in diesem Fall direkt 2 neue Prozeduren; diese werden i.d.R. am Ende unseres Codebereichs eingefügt.

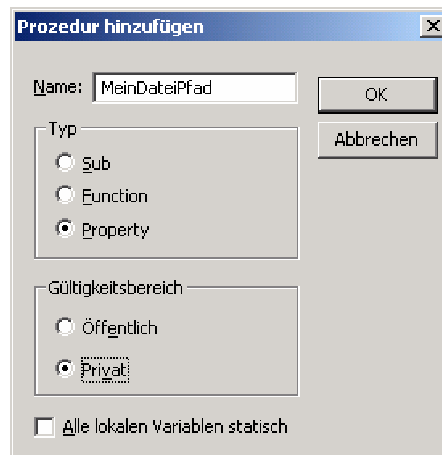


Abbildung 29 - Einstellung einer Property

```
Private Property Get MeinDateiPfad() As Variant

End Property

Private Property Let MeinDateiPfad(ByVal vNewValue As Variant)

End Property
```

Etwas unschön dabei ist die automatische Vergabe des Datentyps Variant für beide Prozeduren. Es ist zu empfehlen hier „typsicher" zu bleiben und jeweils den Datentyp in String zu ändern.

Diese beiden Prozeduren haben jetzt ausschließlich die Aufgabe, sich um die Variable **sMeinDateiPfad** zu kümmern.

Die Let-Prozedur ist dafür zuständig, der Variablen sMeinDateiPfad einen Wert zuzuweisen; und zwar genau den Wert, der im Argument **vNewValue** übergeben wird.

Die Get-Prozedur wiederum ist dafür zuständig, den Wert der Variablen **sMeinDateiPfad** zurückzugeben.

```
Option Explicit

Dim sMeinDateiPfad As String

Private Property Get MeinDateiPfad() As String
    MeinDateiPfad = sMeinDateiPfad
End Property

Private Property Let MeinDateiPfad(ByVal vNewValue As String)
    sMeinDateiPfad = vNewValue
End Property
```

Abbildung 30 - Die entsprechenden Codezeilen

Wir haben jetzt den Zugriff auf die Variable durch entsprechende Property-Prozeduren gekapselt. Der Zugriff erfolgt entsprechend über deren Bezeichnung:

```
Sub Main()
    MeinDateiPfad = "C:\MeineDatei.txt"
    MsgBox "Der aktuelle Dateipfad lautet: " & MeinDateiPfad
End Sub
```

Abbildung 31 - Zugriff auf die Property

Wo liegt denn jetzt der Vorteil?

Der Vorteil liegt in der Möglichkeit, die Variable **vor** Rückgabe bzw. beim „Setzen" zu plausibilisieren. Wenn man den Wert einer Variablen ermittelt, besteht keine Möglichkeit mehr, noch „korrigierend" eingreifen zu können. Bei der Kapselung einer Variablen über Property-Prozeduren ist das jedoch sehr leicht möglich. Darüber hinaus lässt sich ggf. sogar ein „Schreibschutz" einrichten. Für unser Beispiel möchten wir plausibilisieren, ob die Variable bereits initialisiert wurde, und falls ja, ob der Wert der Variable auch gültig ist. Dies stellt sich beispielhaft wie folgt dar:

```
Option Explicit

Dim sMeinDateiPfad As String

Sub Main()

    MeinDateiPfad = "C:\MeineDatei.txt"
    Schreibe_in_die_Datei

End Sub

Sub Schreibe_in_die_Datei()

    Dim i As Integer
    Open MeinDateiPfad For Input As #i
    '...
    Close #i

End Sub

Private Property Get MeinDateiPfad() As String

    MeinDateiPfad = sMeinDateiPfad

End Property

Private Property Let MeinDateiPfad(ByVal vNewValue As String)

   'Bevor wir jetzt die Variable mit dem übergebenen Wert
   'beschreiben, prüfen wir nach, ob die Datei überhaupt vorhanden
   'ist. Falls nicht, setzen wir einen anderen Pfad.
    If Dir(vNewValue) = "" Then
        vNewValue = "C:\MeineDatei2.txt"
    End If
    sMeinDateiPfad = vNewValue

End Property
```

Abbildung 32 - Beispiel-Code

In der hier gezeigten Let-Prozedur wird der ursprünglich vorgesehene Dateipfad verändert.

Diese Plausibilisierung ließe sich natürlich auch in der Get-Anweisung **vor** Rückgabe des Variablenwertes durchführen.

Wann, wo und wie mithilfe dieser Property-Prozeduren Variablenwerte plausibilisiert und ggf. verändert werden ist davon abhängig, welche „Fachlichkeit“ und welche Prüfungen durchzuführen sind.

Grundsätzlich wird der eigene Code durch Nutzung von Property-Prozeduren bei der Bearbeitung von globalen Variablen deutlich sicherer und flexibler.

Nur Wiederholungen mit Schleifen

Eine Schleife verwendet man, um denselben Codeblock einer Kontrollstruktur mehr als einmal auszuführen. Die Wiederholung einzelner oder mehrerer Anweisungen gehört zu den grundlegenden Aufgaben bei der Ausführung vieler mathematischer Berechnungen, dem Herausziehen kleinerer Dateneinträge aus größeren und der Wiederholung einer Aktion an mehreren Mitgliedern einer Gruppe. VBA bietet folgende Schleifenkonstrukte an:

Schleifentyp	Wie sie schleift
`Do...Loop`	Während oder bis eine Bedingung `True` ergibt
`While...Wend`	Während oder bis eine Bedingung `True` ergibt
`For...Next`	Festgelegte Anzahl von Malen
`For Each...Next`	Für jedes Objekt einer „Collection“

Wenn man mit verschachtelten Schleifen arbeitet, muss man daran denken, dass die innere Schleife vor der äußeren Schleife endet!

Do-Schleifen

Die verschiedenen Versionen von **Do...Loop**-Anweisungen sind alle so angelegt, dass sie einen Codeblock so lange ausführen, bis eine Bedingung erfüllt wird. Um zu entscheiden, ob die Schleife weiter ausgeführt werden soll, wertet die **Do...Loop**-Anweisung einen Bedingungsausdruck aus.

Es gibt unzählige Anwendungsmöglichkeiten für **Do...Loop**-Strukturen. Man kann zum Beispiel:

- Eine Fehlermeldung immer wieder einblenden, bis der Mensch vor dem Bildschirm endlich einen gültigen Eintrag in ein Dialogfenster tippt.
- Daten aus einer Datei lesen, bis das Ende der Datei erreicht ist.
- Kurze Strings in langen Strings suchen und zählen.
- Dem Programm für eine festgelegte Zeit Muße gönnen.
- Aktionen an allen Elementen in einem Array vornehmen.
- Mit If...Then-Anweisungen Aktionen mit mehreren Elementen eines Arrays durchführen, die bestimmte Kriterien erfüllen.

Typische Do...Loop-Anweisungen

Anweisung	Ausführung	K/F
`Do While...Loop`	Beginnt und wiederholt den Block nur, wenn die Bedingung wahr ist.	K
`Do...Loop While`	Führt den Block einmal aus und wiederholt ihn, solange die Bedingung wahr ist.	F
`Do Until...Loop`	Beginnt und wiederholt den Block nur, wenn die Bedingung falsch ist.	K
`Do...Loop Until`	Führt den Block einmal aus und wiederholt ihn, solange die Bedingung falsch ist.	F
`Do...Loop`	Wiederholt den Block unbegrenzt lange und macht einen Abgang, wenn ein Bedingungsausdruck innerhalb einer Schleife eine `End Do`-Anweisung ausführt.	-

Hinweis: die **While...Wend**-Anweisung entspricht im Wesentlichen der **Do While**-Anweisung und wird daher nicht separat betrachtet.

Kopf oder Fuß – die Bedingung diktiert den Ablauf

Der Unterschied zwischen **Do While...Loop** und **Do...Loop While**-Anweisungen ist schnell erklärt: Die **Do While...Loop**-Anweisung hat ihrer Bedingung am Anfang der Schleife (kopfgesteuert), während sie bei **Do...Loop While**-Anweisungen am Ende steht (fußgesteuert).

Wann eine kopf- und wann eine fußgesteuerte Schleife angebracht ist, hängt von der Aufgabe ab.

Wenn die Ausführung einer Schleife von einer Bedingung abhängig ist, sollte man diese auch zu Beginn der Schleife überprüfen.

Wenn die Ausführung der Schleife in jedem Fall erfolgen muss und man sicher ist, dass die Bedingung zum Beenden der Schleife noch nicht erreicht ist, kann man die Bedingung auch erst im „Schleifenfuß“ überprüfen.

Ein Beispiel ohne nennenswerten Nutzen:

```
Sub Dreh_die_Zahlen_um()

    Dim sRückwärtsNummer    As String
    Dim lOriginalNummer     As Long
    Dim lEineZahl           As Long

   'Die erste Schleife
    Do While lOriginalNummer < 10
        lOriginalNummer = InputBox("Bitte gibt ein Zahl größer 9 ein!")
    Loop
   'Die zweite Schleife
    Do While lOriginalNummer
        lEineZahl = lOriginalNummer Mod 10
        sRückwärtsNummer = sRückwärtsNummer & Str(lEineZahl)
        lOriginalNummer = Int(lOriginalNummer / 10)
    Loop

    MsgBox sRückwärtsNummer

End Sub
```

Abbildung 33 - Bedingung diktiert den Ablauf

Bitte das Beispiel einfach abtippen und ausprobieren...

Welches Ergebnis erhalten wir bei der Eingabe:

```
617923            ?
987654321123456789 ?
```

Wann man Do ohne While und Until verwendet

Mit Standard **Do...While/Until...Loop**-Anweisungen kann man Bedingungen am Anfang oder am Ende einer Schleife abfragen. Was aber, wenn man das irgendwo mitten in der Schleife tun möchte?

In diesem Fall benutzt man am besten eine **Do...Loop**-Anweisung ohne While oder **Until**. Diese Arbeitstechnik verlangt, dass eine **If**- oder **Select Case**-Anweisung in der Schleife ver-

schachtelt wird. Dabei schließen ein- oder mehrere Zweige der verschachtelten Bedingungsanweisung eine **Exit Do**-Anweisung ein, die dem Programm erlaubt, die Schleife zu beenden, wenn eine festgelegte Bedingung erfüllt wird.

```
Do
   'bei jedem Durchlauf der Schleife ausführende Anweisungen:
    If Bedingung Then
        Exit Do
    End If
   'weitere auszuführende Anweisungen, falls Schleife weiterläuft
Loop
```

Abbildung 34 - Do...Loop Anweisung

Man sieht hier recht deutlich, dass sich dieses Vorgehen gut eignet, wenn man einige der Anweisungen der Schleife unabhängig von der Erfüllung der Bedingung ausführen möchte. Es ist darüber hinaus nützlich, wenn die Schleife unter mehreren verschiedenen Bedingungen enden soll.

Noch ein Beispiel ohne nennenswerten Nutzen:

```
Sub Antwort_bekommen()

    Dim sAntwort As String

    sAntwort = InputBox("Gib Deine Antwort (A-E) ein.")
    Do
        If sAntwort = "" Then
            sAntwort = InputBox("Du hast nichts eingeben." & vbCrLf & _
                                "Bitte gib einen Buchstaben zwischen A und E ein.")

        ElseIf Len(sAntwort) > 1 Then
            sAntwort = InputBox("Die Antwort sollte nur 1 Buchstabe sein." & vbCrLf & _
                                "Versuche es noch einmal.")

        ElseIf UCase(sAntwort) < "A" Or UCase(sAntwort) > "E" Then
            sAntwort = InputBox("Du hast ein ungültiges Zeichen eingegeben." & vbCrLf & _
                                "Gib einen Buchstaben zwischen A und E ein.")

        Else

            Exit Do
        End If
    Loop

    MsgBox "Na geht doch ;o)"

End Sub
```

Abbildung 35 - Do...Loop ein weiteres Beispiel

Abzählen mit For...Next-Schleifen

Wenn wir bereits vor dem Programmstart wissen, wie oft eine Schleife ausgeführt werden soll, verwendet man am besten einer **For...Next**-Schleife. Durch die Eingabe von Start- und End-Werten, bei denen es sich um ganze Zahlen, Variablen oder gar komplexen Ausdrücken handeln kann, legt man fest, wie oft VBA die Schleife durchlaufen soll. Währen die Schleife läuft, registriert eine Zähler-Variable die Anzahl vollendeten Durchläufe. Sobald der Wert des Zählers mit dem End-Wert übereinstimmt, endet die Schleife.

Vereinfacht sieht die Syntax einer **For...Next**-Struktur so aus:

```
For Zähler = Startwert To Endewert
    '(bei jedem Durchlaufder Schleife auszuführende Anweisung)
Next Zähler
```

Abbildung 36 - For...Next Anweisung

oder als Beispiel, bei dem auch etwas „passiert“:

```
Dim i As Integer
For i = 1 To 10
    'Ausgabe des aktuellen Wertes im Direktfenster
     Debug.Print "Das ist Durchlauf " & Str(i)
Next i
```

Abbildung 37 - For...Next mit Ausgabe

In diesem Beispiel sind die Werte für Start und Ende jeweils Zahlen. Wenn die Schleife beginnt, wird die Variable **i** auf 1 gesetzt – mit anderen Worten: Der Wert von Start wird der Zählervariablen zugeordnet. Nach jedem kompletten Durchlauf erhöht **Next i** den Wert von **i** um 1, und die Schleife beginnt von vorne. Wenn der Wert von **i** 10 erreicht, endet die Schleife.

Verschachtelte For...Next Schleifen

Wie andere VBA-Strukturen auch, können For...Next-Schleifen ineinander – oder in andere Kontrollstrukturen – verschachtelt werden, und zwar so oft man will. Der folgende Codeblock verdeutlicht das Prinzip (verbunden mit der Frage, was passiert da eigentlich?):

```
Sub GoodLuck()

    Dim i As Integer
    Dim j As Integer
    Dim k As Integer
    Dim s As String
    Dim t As String

    For i = 1 To 6
        For j = 1 To 6
            Do
                k = Int((49 * Rnd) + 1)
            Loop Until InStr(s, Str(k)) = 0
            If Len(s) > 0 Then
                s = s & vbTab
            End If
            s = s & Str(k)
        Next j
        t = t & s & vbCrLf
        s = ""
    Next i
    MsgBox t

End Sub
```

Abbildung 38 - Verschachtelte For...Next Schleife

Kapitel

5

ABBILDUNGSVERZEICHNIS

Abbildung 1 - Die DIE 5
Abbildung 2 - Eine UserForm....9
Abbildung 3 - Die Werkzeugsammlung....9
Abbildung 4 - Ein Bezeichnungsfeld einfügen 10
Abbildung 5 - Das Eigenschaften-Fenster 11
Abbildung 6 - Die Befehlsschaltfläche hinzufügen 12
Abbildung 7 - Das bisherige Ergebnis 13
Abbildung 8 - Die Grundstruktur 14
Abbildung 9 - Die fertige Prozedur 14
Abbildung 10 - Elemente in der Objektliste 15
Abbildung 11 - Elemente in der Ereignisliste 15
Abbildung 12 - Die neuen Codezeilen....16
Abbildung 13 - Das angepasste Formular 18
Abbildung 14 - ASCII-Tabelle....20
Abbildung 15 - Die fertige Routine 21
Abbildung 16 - Die veränderten Ereignis-Routinen....23
Abbildung 17 - Auszug aus der OL-Hilfe: LEN-Funktion....24
Abbildung 18 - Auszug aus der OL-Hilfe: MsgBox-Funktion....24
Abbildung 19 - Die Plausibilität der Datumsprüfung....25
Abbildung 20 - Die veränderte Routine 27
Abbildung 21 - Der Grundaufbau unserer Prozedur Alter 27
Abbildung 22 - Die "einfache" Altersberechnung 28
Abbildung 23 - Die Profi-Altersberechnung 28
Abbildung 24 – Musterlösung: Die fertigen Routinen Teil 1....30
Abbildung 25 - Musterlösung: Die fertigen Routinen Teil 2 31
Abbildung 26 - Musterlösung: Der Dialog....32
Abbildung 27 - OL-Hilfe zum Thema Prozedur 33
Abbildung 28 - Code-Beispiel 34
Abbildung 29 - Einstellung einer Property....36
Abbildung 30 - Die entsprechenden Codezeilen 37
Abbildung 31 - Zugriff auf die Property 37
Abbildung 32 - Beispiel-Code 38
Abbildung 33 - Bedingung diktiert den Ablauf....40
Abbildung 34 - Do...Loop Anweisung....41
Abbildung 35 - Do...Loop ein weiteres Beispiel....41
Abbildung 36 - For...Next Anweisung 42
Abbildung 37 - For...Next mit Ausgabe....42
Abbildung 38 - Verschachtelte For...Next Schleife....42