fu:stat

Makroprogrammierung in Excel mit VBA

Dominik Vogt

Kontakt:
Dominik.Vogt@fu-berlin.de

VORBEMERKUNG

Im Kurs soll die Makroprogrammierung mit VBA erlernt werden. Das Skript folgt im Aufbau aber nicht im Detail dem Kurs. Was im Kurs exemplarisch gezeigt wird, soll hier eher systematisch dargestellt werden. Dies gilt insbesondere für die Teile zur Programmiersprache VBA; sie sollen dem/r Kursteilnehmer/in nachträglich erlauben das Wichtige nachzulesen.

Das Skript gliedert sich in vier Teile: Nach einer Einführung in die Makrowerkzeuge und die VBA-Entwicklungsumgebung (Kapitel 1) wird gezeigt, wie aufgezeichnete Makros bearbeitet werden können (Kapitel 2). Danach werden die für die Makroprogrammierung wichtigen VBA-Sprachelemente dargestellt (Kapitel 3). Die Darstellung beschränkt sich in vielen Fällen auf die bloße Auflistung der Methoden und Eigenschaften. Mit den entsprechenden Schlüsselwörtern können Sie leicht in der Hilfe detaillierte Informationen suchen. Im letzten Kapitel (Kapitel 4) wird an einem ausführlichen Beispiel die Anwendung des zuvor Erläuterten gezeigt.

Unterlagen:

http://userpage.fu-berlin.de/~vodo/vbakurs/index.html

INHALT

1 EINFÜHRUNG

1.1 MAKROS UND VBA

Makro: Ein Makro ist eine Folge von Befehlen, die unter einem Namen zusammengefasst werden. Sie dienen der Automatisierung häufig verwendeter Befehlsfolgen (oder Arbeitsschritten).

In Excel kann die gewünschte Befehlsfolge einfach aufgezeichnet werden. Die Befehle für ein Makro können aber auch selbst geschrieben (bzw. programmiert) werden. Die Programmiersprache für Makros in Excel und anderen Office-Anwendungen ist VBA. Bei der Aufzeichnung von Makros erstellt Excel automatisch den Programmcode.

VBA: VBA steht für *Visual Basic for Application*. VBA basiert auf der Programmiersprache *Visual Basic* und ergänzt diese um Befehle und Elemente, mit denen bestimmte Anwendungen - in unserem Falle Excel - gesteuert werden können.

Einige Hintergründe: Visual Basic ist eine von Microsoft aus BASIC entwickelte Programmiersprache. Von Visual Basic gibt es zurzeit zwei Varianten: Das neuere, vollständig objektorientierte VB.NET (auf dem .NET-Framework basierend) und eine "klassische" nur teilweise objektorientierte Version (bis VB 6). VBA basiert auf der klassischen Variante. Visual Basic ist als Dialekt von BASIC verhältnismässig einfach zu lernen, steht BASIC doch für: **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode.

EINIGE WICHTIGE BEGRIFFE

Objektorientierung: Unter Objektorientierung versteht man ein bestimmtes Konzept der Programmierung. Ein Programm wird dabei nicht als Abfolge von Anweisungen oder Befehlen verstanden (wie beim imperativen oder prozeduralen Programmieren), sondern als das Definieren von Klassen und das Verwenden dieser Klassen als sog. Objekte mit definierten Eigenschaften und Fähigkeiten (sog. Methoden). Eine Klasse ist gewissermaßen die Idee oder der Bauplan eines Objektes; Verwirklichung der Klasse, das konkrete Objekt nennt man Instanz.

Für die Makroprogrammierung muss man sich nicht mit der Definition von Klassen auseinandersetzen. In VBA für Excel sind die Klassen für viele Dinge in Excel definiert, die Sie als Objekte in Makros verwenden können. Dies macht das "for Application" aus. So ist etwa für ein Arbeitsblatt die Klasse WORKSHEET definiert. VBA gibt es auch für Word oder für PowerPoint.

VBA übernimmt von Visual Basic Schlüsselwörter, Anweisungen und die Syntax. Die **Syntax** einer Programmiersprache beschreibt ähnlich derjenigen einer natürlichen Sprache, wie

Anweisungen geschrieben werden müssen. Eine Folge von Anweisungen wird **Code** genannt. Code kann ein Teil oder ein ganzes Programm sein. **Schlüsselwörter** sind reservierte Zeichen oder Wörter (Zeichenketten), mit denen die grundlegenden Befehle geschrieben werden. Solche grundlegenden Befehle sind bspw. die Definition von Funktionen und Methoden, Deklarationen von Variablen und Wertezuweisungen oder Anweisungen für den Kontrollfluss, d.i. die Reihenfolge mit der Befehle abgearbeitet werden.

Eine **Funktion** oder eine **Methode** ist eine eigenständige, kleinere oder grössere Abfolge von Anweisungen, die durch eine andere Funktion oder Methode aufgerufen werden kann. Im Unterschied zur Methode gibt die Funktion einen Wert zurück. In Excel können Methoden als Makros und Funktionen als Tabellenfunktionen verwendet werden.

1.2 MAKROWERKZEUGE

Excel bietet spezielle Werkzeuge, um Makros zu erstellen und auszuführen. Dazu zählen:

- das Dialogfenster, um Makros auszuführen
- die Makroaufzeichnung
- der Visual-Basic Editor oder VBA-Entwicklungsumgebung
- das Makrosicherheitscenter
- der Entwurfsmodus und die Steuerelemente

Zusammengefasst sind diese Werkzeuge im Menüband auf dem Register "Entwicklungstools". Standardmässig ist das Register nicht sichtbar und muss über "Menüband anpassen" (Rechtsklick aufs Menüband oder unter Optionen) aktiviert werden:

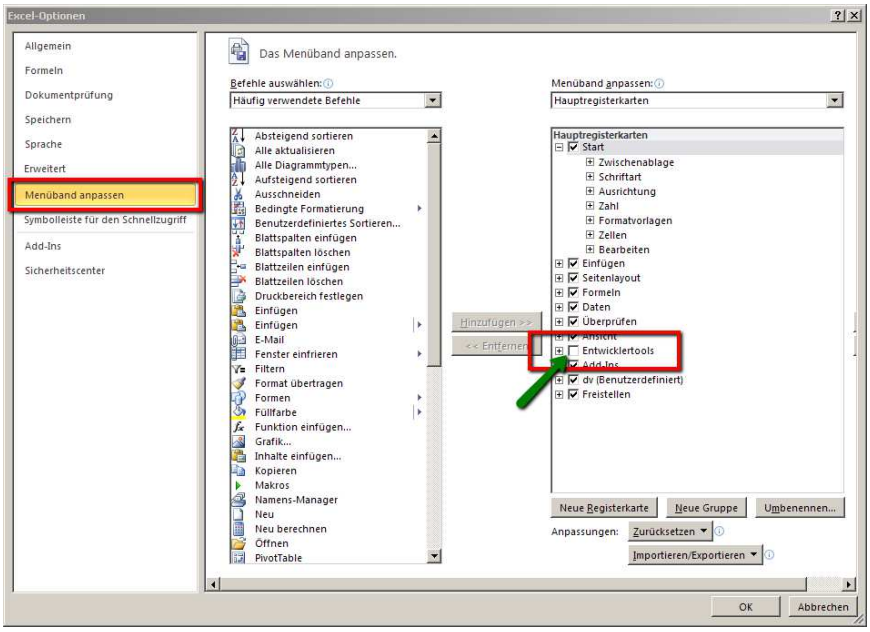


Abbildung 1: Entwicklungstools in den Optionen aktivieren

Auf dem Register "Entwicklungstools" befinden sich die Befehle in der Gruppe "Code":

Abbildung 2: Register "Entwicklungstools"

Die Makroaufzeichnung kann man auch direkt in der Statusbar starten (evtl. deaktiviert):

Abbildung 3: Aufzeichnung starten

und stoppen:

Abbildung 4: Aufzeichnung stoppen

Makro-Dialog: Das wichtigste Makrowerkzeug ist ein kleines Fenster, das ich den Makro-Dialog nennen werde. Der Makro-Dialog kann mit dem Shortcut ALT+F8 oder der "Makros"-Schaltfläche auf dem Register "Entwicklungstools" gestartet werden: 

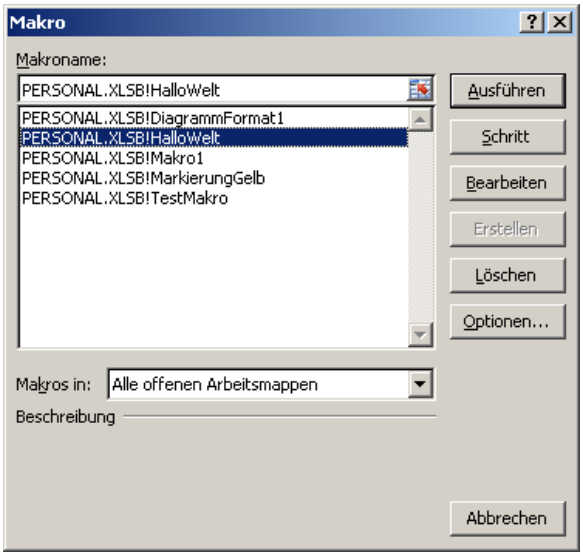


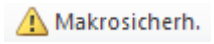
Abbildung 5: Makro-Dialog

Der Makro-Dialog dient hauptsächlich dazu, Makros auszuführen – Befehl: "Ausführen". Er kann aber auch verwendet werden um Makros zu verwalten: Unter Optionen kann man Makros eine Beschreibung oder ein Tastenkürzel zuordnen. Mit "Löschen" können Makros wieder entfernt werden.

Der Makro-Dialog bietet aber auch Zugang auf die VBA-Entwicklungsumgebung. "Bearbeiten" öffnet den VBA-Editor beim ausgewählten Makro. Mit "Erstellen" wird im VBA-Editor ein neues Makro mit dem eingegebenen Namen erstellt.
Der Befehl "Schritt" startet den sog. Debugging-Modus. Als "Debugging" bezeichnet man das Suchen von Fehlern in einem Programm. Der Debugging-Modus erlaubt es jeden Befehl eines Makros einzeln, eben Schritt für Schritt, auszuführen und den Effekt zu beobachten.

VBA-Editor: Für die Makro-Programmierung selbst steht die sog. VBA-Entwicklungsumgebung oder der VBA-Editor zur Verfügung. Der VBA-Editor ist ein eigenständiges Programm, um VBA-Code zu schreiben und zu korrigieren ("debuggen"). Er stellt eine Vielzahl von Werkzeugen zur Verfügung, die wir teilweise kennenlernen werden. Wichtig ist die Hilfe der VBA-Entwicklungsumgebung, denn Informationen zur Programmiersprache und eine Referenz der VBA-Objekte von Excel finden sich dort und nicht in den Excel-Hilfsdateien.
Der VBA-Editor wird mit ALT-F11 oder mit der Schaltfläche "Visual Basic" auf dem Register "Entwicklungstools"  gestartet.

1.3 MAKROSICHERHEIT

Bevor man mit Makros arbeiten kann, muss man sicherstellen, dass Makros überhaupt ausgeführt werden können. Im Makrosicherheitscenter 

(unter Entwicklungstools/Code/Makrosicherh.) können Makros aktiviert oder deaktiviert werden.

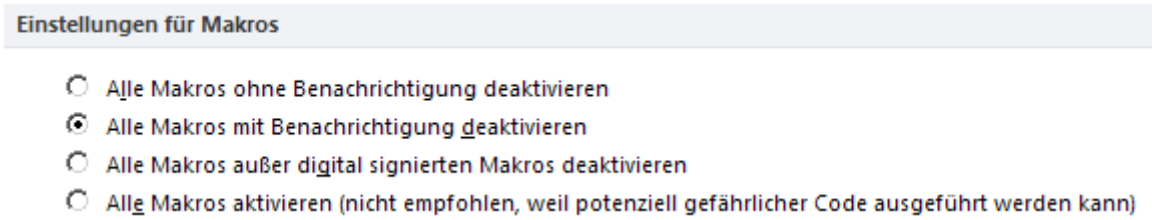


Abbildung 6: Makrosicherheitscenter

Im Normalfall empfiehlt sich die Einstellung, bei der Excel jeweils nachfragt, ob die Makros eines Dokumentes aktiviert werden sollen. Es erscheint eine gelbe Leiste, in der Sie durch einen Klick auf "Inhalte aktivieren" die Makros aktivieren können:

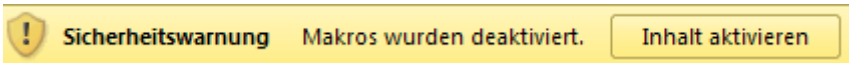
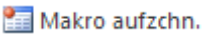


Abbildung 7: Sicherheitswarnung

2 ERSTE SCHRITTE

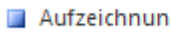
2.1 AUFZEICHNEN MAKROS

Die einfachste Methode ein Makro zu erstellen, ist das Aufzeichnen einer Abfolge von Aktionen. Sie können damit alles, was Sie in Excel machen können, automatisieren.

Aufzeichnen: Das Aufzeichnen von Makros ist an sich einfach. Mit einem Klick auf 

öffnet sich ein kleiner Dialog (alle Befehle befinden sich auf dem Register "Entwicklertools" in der Gruppe "Code"). Hier müssen Sie dem Makro einen Namen geben und können den Speicherort festlegen (zum Speichern von Makros vgl. nächstes Kapitel).

Indem der "Makro aufzeichnen"-Dialog mit OK beendet wird, beginnt die Aufzeichnung. Alles was Sie jetzt tun, wird wiederholt, wenn Sie das Makro ausführen.

Anstelle des Makro-Aufzeichnen Befehls erscheint während der Aufzeichnung der Befehl, um diese zu beenden: 

. Der Beenden-Befehl erscheint auch in der Statuszeile (muss evtl. über das Kontextmenü aktiviert werden).

Excel zeichnet immer ausgehend vom aktuell ausgewählten oder aktiven Objekt auf. Wenn Sie das Makro danach ausführen, geht Excel vom aktiven Objekt aus. – Eine Schwierigkeit ist, dass Excel auch aufzeichnet, wenn ein Objekt ausgewählt wird. Beim Abspielen wird dann wiederum genau dieses Objekt ausgewählt. Dies kann zu unerwünschten Ergebnissen führen.

Relative Aufzeichnung: Während Excel in der Regel die absolute Adresse von Zellen aufzeichnet, wird bei der relativen Aufzeichnung die Position bezüglich der letzten aktiven Zelle gespeichert. Zeichnen Sie bspw. ein Makro in A1 auf und verschieben Sie den Cursor nach B3, wird beim Abspielen in z.B. C2 der Cursor auf D4 gesetzt. Die relative Aufzeichnung können Sie vor oder während dem Aufzeichnen eines Makros mit der Schaltfläche 

ein- und ausstellen.

2.2 AUSFÜHREN MAKROS

Ausführen: Verwenden können Sie jedes Makro über den Makro-Ausführen Dialog. Gestartet wird dieser über den Befehl "Makros":  . Oder mit der Tastenkombination ALT+F8. Im Makro-Dialog müssen Sie das gewünschte Makro auswählen und dann über den Befehl "Ausführen" starten. Ist ein Makros nicht in der Liste vorhanden, dann müssen Sie im DropDown "Makros in:" den entsprechenden Speicherort wählen.

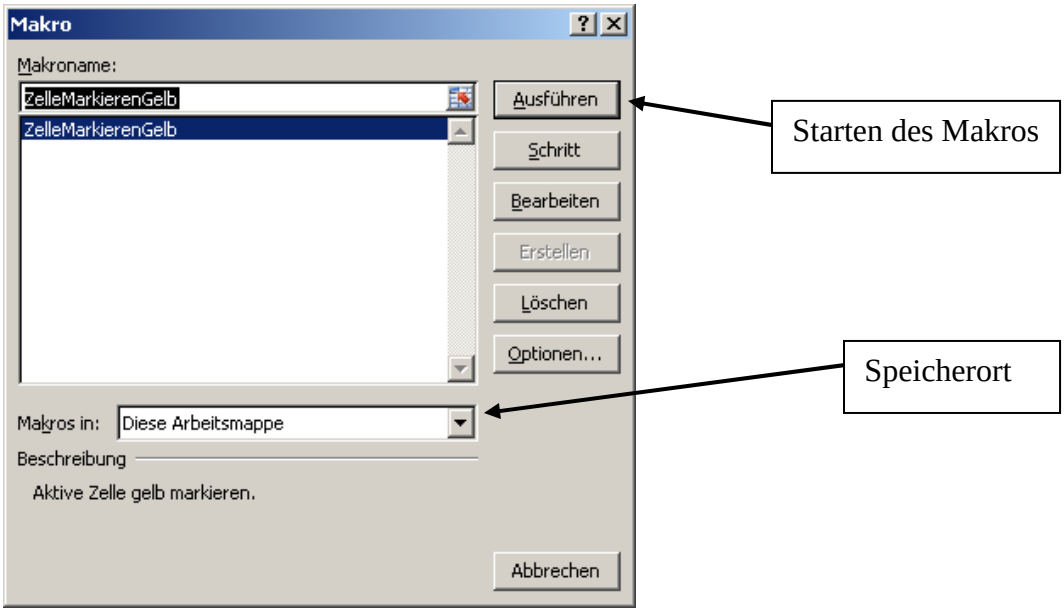


Abbildung 8: Makro-Dialog

Wichtiger Hinweis: Die Arbeitschritte, die durch ein Makro ausgeführt werden, können *nicht rückgängig* gemacht werden. Wird durch ein Makro Inhalt überschrieben oder gelöscht, ist dieser verloren. Mehr noch: manuelle Aktionen, die vor dem Ausführen eines Makros ausgeführt wurden, können danach auch nicht mehr rückgängig gemacht werden (es sei denn, diese betreffen ein anderes Tabellenblatt).

Verwenden von Makros: Neben dem Makro-Dialog können Makros auch über das Menüband (über "Menüband anpassen", Kategorie: "Makros") oder mittels Tastenkombinationen (im Makro-Dialog definieren) ausgeführt werden. Zusätzlich können Makros mit Zeichenobjekten (Tabellen, Textfelder oder Schaltflächen) verknüpft werden (Kontextmenü des Objektes: "Makro zuweisen").

2.3 SPEICHERN VON MAKROS

Makros können in einer Arbeitsmappe, einem Add-In oder in der sog. Persönlichen Arbeitsmappe gespeichert werden. Das DropDown "Makro speichern in" des Dialogs zum Aufzeichnen von Makros bietet verschiedene Speicheroptionen.

Makroarbeitsmappe: Makros können Teil einer Arbeitsmappe sein. Sie stehen dann überall mit dieser zur Verfügung. Um in der Arbeitsmappe Makros zu speichern, müssen Sie diese als "Excel-Arbeitsmappe mit Makros (*.xlsm)" oder als "Binärarbeitsmappe (*.xlsb)" abspeichern. Das normale Excel-Arbeitsmappenformat (Endung .xlsx) speichert keine Makros.

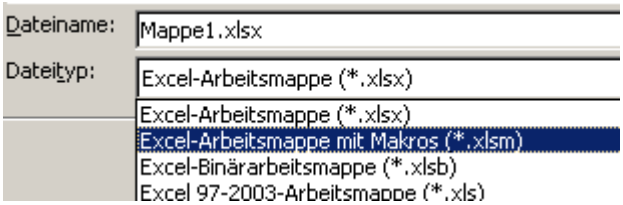


Abbildung 9: Dateityp-Auswahl des Speichern-unter-Dialogs

Persönliche Makroarbeitsmappe: Im Unterschied zu einer normalen Arbeitsmappe hat die Persönliche Makroarbeitsmappe den Vorteil, dass die Makros automatisch mit Excel zur Verfügung stehen. In der Regel ist sie der beste Platz, um Makros zu speichern.
An sich ist die Persönliche Makroarbeitsmappe eine normale Arbeitsmappe, Excel legt sie unter dem Namen "PERSONAL.XLSB" im sogenannten Startup-Ordner an. Alle Excel-Dateien in diesem Ordner werden beim Start von Excel automatisch geöffnet. Der Standardpfad für den Startup-Ordner ist:
XP: C:\Dokumente und Einstellungen\Benutzername\Anwendungsdaten\Microsoft\Excel\XLSTART
Windows 7: C:\Users\Benutzername\AppData\Roaming\Microsoft\Excel\XLSTART
Unter Optionen/Erweitert/Allgemein kann man auch einen eigenen Start-Ordner angeben.
Die Persönliche Arbeitsmappe wird geöffnet, aber nicht angezeigt. Man könnte Sie über den Reiter "Ansicht", Gruppe "Fenster" mit dem Befehl "Einblenden" anzeigen lassen oder wieder ausblenden.

Hinweis: Wenn Sie für ein Makro eine Schaltfläche auf dem Menüband einrichten (Menüband anpassen), dann speichert Excel dabei auch den Pfad auf die Arbeitsmappe. Sie können damit also auf Makros in einer beliebigen Arbeitsmappe zugreifen.

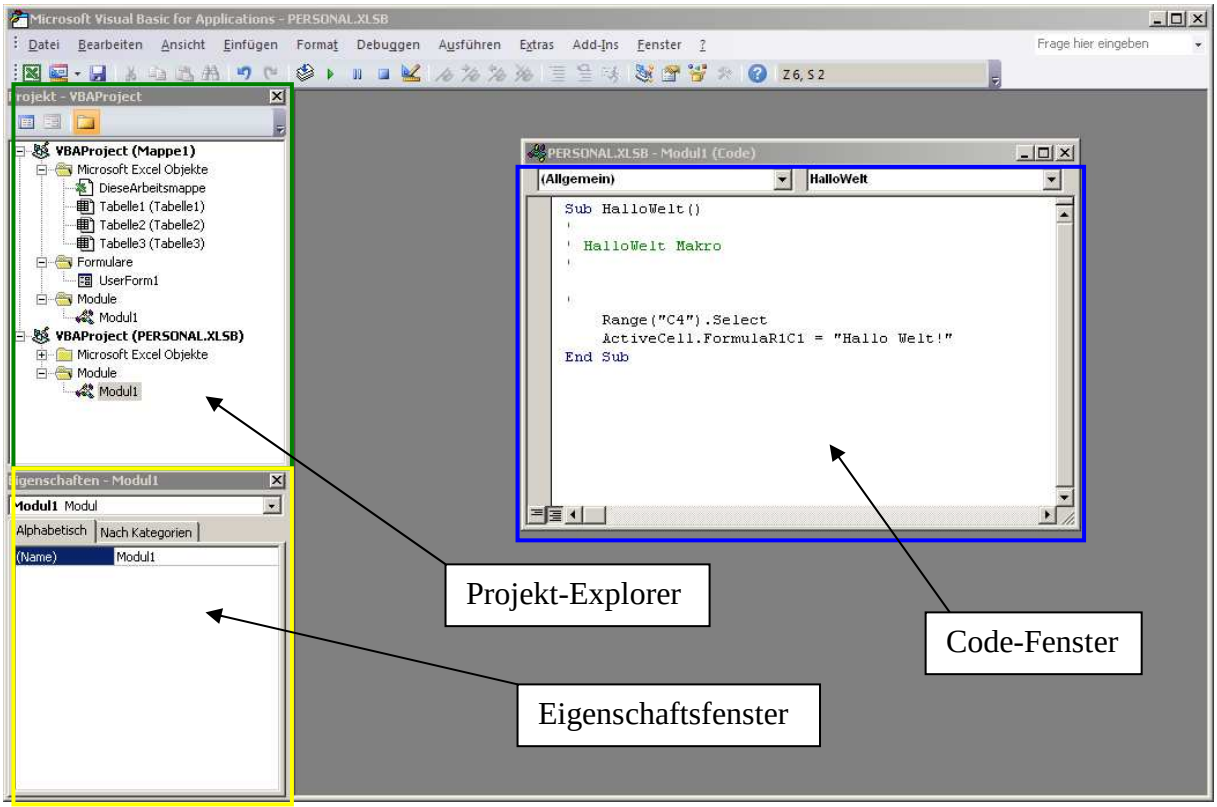
Add-In: In Excel können Makros auch in sog. Add-Ins abgespeichert werden. Makros, die in Add-Ins abgespeichert sind, erscheinen nicht im Makro-Dialog, Sie können diese aber im Menüband einbinden, oder als Arbeitsblatt-Funktionen verwenden. Der Vorteil ist, dass Sie bei Funktionen den Arbeitsmappennamen nicht angeben müssen.
Erstellen eines Add-Ins: Um ein Add-In zu erstellen, müssen Sie eine Arbeitsmappe mit dem Dateityp "Excel-Add-In (*.xlam)" abspeichern (ganz unten in der Liste). Excel wechselt dann direkt in den Standardspeicherort für Add-Ins. Man kann Add-Ins aber auch in einem anderen Ordner abspeichern.
Einbinden eines Add-Ins: Man kann ein Add-In einfach öffnen und dann verwenden. Will man die Makros aber automatisch beim Start von Excel zur Verfügung haben, muss man das

Add-In einbinden. Hierzu müssen Sie in den Excel-Optionen unter "Add-Ins" bei "Verwalten" die "Excel-Add-Ins" auswählen und mit "Gehe zu" den "Add-In"-Dialog öffnen. Nachdem hier ein Haken gesetzt wurde, stehen die Makros zur Verfügung. Wenn Ihr Add-In nicht in der Liste erscheint, können Sie dieses mit "Durchsuchen" hinzufügen.

2.4 Anpassen von Makros und VBA-Entwicklungsumgebung

Der einfachste Einstieg in die Makroprogrammierung ist das Anpassen eines aufgezeichneten Makros. Beim Aufzeichnen des Makros erstellt Excel eine VBA-Methode und schreibt jede ausgeführte Aktion als eine Zeile VBA-Code, als eine "Anweisung" oder auch Befehl. Sie können das nutzen, um gewissermaßen passiv die Programmiersprache VBA zu lernen. Man lernt so nicht die Syntax von VBA, aber die Excel spezifischen Namen von Objekten und Methoden. Selbst als erfahrener VBA-Programmierer zeichne ich oft noch Makros auf, wenn ich den Namen eines Objektes oder einer Methode nicht kenne.

Zeichnen wir ein einfaches "HalloWelt"-Makro auf. Starten Sie auf einem leeren Arbeitsblatt die Aufzeichnung, setzen Sie den Cursor in die Zelle C4 und geben Sie den Text "Hallo Welt" ein. Beenden Sie die Aufzeichnung und starten sie den Makro-Dialog. Wählen Sie das "HalloWelt"-Makro aus und klicken Sie auf "Bearbeiten" (das funktioniert dummerweise nur, wenn man das Makro in der "Aktuellen Arbeitsmappe" gespeichert hat; geht es nicht klicken Sie stattdessen auf "Schritt"). Der VBA-Editor öffnet sich und sollte etwas so aussehen:



Code-Fenster: Das Makro, das wir soeben aufgezeichnet haben, sehen Sie im Code-Fenster (blauer Rand). Das Code-Fenster kann auch maximiert sein und füllt dann den ganzen grauen Bereich.

Methode oder Prozedur: Das Makro beginnt mit dem Wort "Sub" gefolgt vom Makronamen und einer leeren Klammer und endet mit der Zeile "End Sub". "Sub" und "End Sub" sind Schlüsselwörter von VBA (blaue Schrift), mit denen eine *Methode* (oder eine Prozedur) definiert wird.

Kommentare: Die folgenden Zeilen in grüner Schrift sind sog. *Kommentare*. Kommentare sind Code-Zeilen, die nicht ausgeführt werden; sie dienen dazu, Programmcode für einen Anderen oder ein späteres Selbst zu erläutern. In VBA werden Kommentare zeilenweise mit einem Hochkomma ' eingeleitet. Es ist gute Programmierpraxis seinen Code ausführlich zu kommentieren.

Anweisungen: Danach folgen zeilenweise - eingerückt und in schwarzer Schrift - die beiden Anweisungen für die zwei Aktionen, die wir aufgezeichnet haben:

Zuerst haben wir die Zelle C4 ausgewählt:

```
Range("C4").Select
```

Danach haben wir in die aktive Zelle den Text "Hallo Welt!" eingegeben:

```
ActiveCell.FormulaR1C1 = "Hallo Welt!"
```

Dies lässt sich leicht nachvollziehen, indem man das Makro schrittweise ausführt. Im Menü "Debuggen" heisst der Befehl dafür "Einzelschritt", bequemer mit dem Shortcut F8. Ein gelber Pfeil und Hintergrund markieren, welche Anweisung als nächstes ausgeführt wird. Die Marke liegt auf "`Sub Hallo Welt()`", das bedeutet die Methode wird betreten. Im Menü "Debuggen" gibt es auch einen Befehl, mit dem man selbst festlegen kann, welche Anweisung im Rahmen der Prozedur als nächstes ausgeführt werden soll: "Nächste Anweisung festlegen". Dies - und die Möglichkeit einen Ausdruck zu überwachen ("Debuggen": "Überwachung hinzufügen") - erlauben es sehr gut rumzuexperimentieren, bis der Code auch wirklich macht, was man will.

Tipp: Die beiden Anweisungen einzurücken ist gute Programmierpraxis. Damit wird markiert, dass sie syntaktisch von den beiden Anweisungen "Sub" und "End Sub", einem Anweisungsblock, eingeschlossen sind, – sie gehören zur Methode, die damit definiert ist.

Modul: Das Makro, genauer die Prozedur "HalloWelt" ist Teil eines sog. *Moduls*. Beim Aufzeichnen des Makros wurde automatisch ein Modul mit dem Namen "Modul1" erstellt. Im *Eigenschaftsfenster* kann man den Namen des Moduls ändern. Ein Modul kann man als das

Dokument verstehen, in welchem Makro-Code geschrieben wird. Ein Modul kann mehrere Methoden oder Makros enthalten. Ein Modul ist selbst Teil einer Arbeitmappe, die man aus VBA-Sicht ein Projekt nennt.

Projekt-Explorer: Im *Projekt-Explorer* sind die geöffneten Projekte mit ihren Elementen sichtbar. Zu einem VBA-Projekt in Excel können auch andere Objekte gehören, wie Klassen oder Formulare. Element der Arbeitsmappe als VBA-Projekt sind immer auch die Arbeitsmappe selbst und ihre Tabellen; in VBA sind diese selbst Objekte, auf deren Methoden und Eigenschaften man im Code zugreifen kann. – Unter dem Menüpunkt "Einfügen" oder mit der Schaltfläche  kann man neue Module oder andere Objekte zum Projekt hinzufügen.
Eigenschaftsfenster: Ein Modul selbst ist gewissermaßen ein primitives Objekt mit nur dem Namen als Eigenschaften. Die Eigenschaft "(Name)" ist der Name oder Bezeichner des Objekts, mit dem es von anderen Objekten aus angesteuert werden kann. Wählt man im Projekt-Explorer eine der Tabellen (Tabelle1) aus, sieht man im Eigenschaftsfenster einige seiner Eigenschaften. Neben dem VBA-Namen besitzt die Tabelle noch eine weitere Eigenschaft "Name" (ohne Klammern). Unter diesem Namen erscheint die Tabelle in Excel auf dem Tabellenreiter (ändern Sie ihn und wechseln Sie zurück zu Excel). – Das Eigenschaftsfenster wird vor allem dann wichtig, wenn Sie eigene Formulare erstellen. Denn ein Formular und die Steuerelemente, die Sie in ihm platzieren können, sind Objekte, deren Eigenschaften Sie im Eigenschaftsfenster direkt (und einfacher als über Anweisungen) festlegen können.

Makro-Anpassen:
Die einfachste Übung wäre jetzt, den Text, den wir in die Zelle eingegeben haben, zu ändern. Hierfür müsste man einfach den Text "Hallo Welt!" ändern. Einen festen Text, oder wie man sagt: einen String, schreibt man in VBA immer in Anführungszeichen.
Oder: Beim Aufzeichnen haben wir irrtümlich die Zelle ausgewählt. Das Ziel war es eigentlich, den Text in die gerade aktive Zelle zu schreiben. – Man müsste hierfür nur die Anweisung löschen oder auskommentieren, die dafür zuständig ist: "`Range("C4").Select".`

Man kann die aufgezeichneten Befehle aber auch so ändern, dass ein Makro entsteht, welches man gar nicht aufzeichnen könnte. Wir wollen den Inhalt der Zelle C4 in die aktive Zelle schreiben.
Analysieren wir die Anweisungen ein wenig:

Die erste Anweisungszeile "`Range("C4").Select`" aktiviert eine Zelle: Von einem Objekt in Excel, nämlich der Zelle C4 alias "`Range("C4")`", rufen wir die Methode auf, die sie zur aktiven Zelle macht (`Select`).

Diese Schreibweise mit einem Punkt ist typisch für objektorientierte Programmiersprachen. Objekte sind Instanzen von Klassen. Innerhalb einer Klasse sind die Eigenschaften und Aktivitäten (Methoden) dieser Objekte definiert. Der Aufruf dieser Komponenten der Klassen erfolgt immer in der Form `Name_der_Instanz.Name_der_Komponente`.

Die Klasse `Range` definiert ein sehr wichtiges Objekt in Excel, nämlich ein Bereich (engl. Range) von Zellen. `Range("C4")` ist gewissermaßen der Name einer Instanz der Klasse, d.i. eines konkreten Objekts im aktiven Arbeitsblatt von Excel. Ich sage "gewissermaßen" denn hier handelt es sich um ein Kürzel. In der objektorientierten Programmierung spiegelt sich genau dieses Abhängigkeitsverhältnis: ein konkreter Zellbereich des aktiven Arbeitsblattes von Excel; in VBA: `Application.ActiveSheet.Range("C4").`

Die zweite Anweisungszeile "`ActiveCell.FormulaR1C1 = "Hallo Welt!"`" weist der aktiven Zelle einen Wert zu. Das gleiche Objekt, nämlich die Zelle C4, sprechen wir jetzt mit dem Namen `ActiveCell` an. Auch `ActiveCell` ist eine Instanz der Klasse `Range`. Und in dieser Klasse ist definiert, dass eine Zelle die Eigenschaft hat, eine Formel als Inhalt haben zu können. Mit dem Gleichheitszeichen weisen wir dieser Eigenschaft einen Wert zu und damit der Zelle einen Inhalt.

Anders als "`Range("C4")`" spricht `ActiveCell` die Zelle nicht über die Adresse an, sondern als die Zelle, die gerade aktiv ist (bei aufgezeichneten Makros oft auch `Selection`), an. Die beiden Instanzen sind nur dann dasselbe konkrete Objekt, wenn der Cursor in C4 ist. Wir können also der Inhaltseigenschaft der aktiven Zelle statt des festen Textes auch den Inhalt der Zelle C4 zuweisen. Beides sind Instanzen der Klasse `Range`, beide haben also die Eigenschaft `FormulaR1C1`.

Die Anweisung für unser neues Makro wäre also:

```
ActiveCell.FormulaR1C1 = Range("C4").FormulaR1C1
```

Mit der Schaltfläche oder mit F5 können Sie die Methode ausführen (der Cursor muss innerhalb der Methode platziert sein).

Tipp: Der VBA-Editor erleichtert das Schreiben von Code durch eine Autovervollständigungsfunktion. Wenn Sie eine Anweisung eintippen, können Sie diese mit Strg-Leertaste automatisch abschließen. Ist die Eingabe nicht eindeutig, dann erscheinen Vorschläge. Mit den Pfeiltasten (auf und ab) oder durch weiteres Tippen können Sie den gewünschten Befehl wählen und mit dem Tabulator vervollständigen. Auch Punkt, Leerzeichen und Enter vervollstän-

digt den Befehl. – Beim Eintippen des Punktes nach dem Namen einer Instanz erscheint eine Auswahl aller Eigenschaften und Methoden der Klasse automatisch.

Tipp: Werte von Variablen oder Objekt-Eigenschaften werden im Debug-Modus als Tooltip angezeigt. Eine Übersicht über alle Variablen einer Prozedur bietet das Local-Fenster. Mit dem Überwachungsfenster können gezielt ausgewählte Variablen beobachtet werden (Menü "Debuggen": "Überwachung hinzufügen"; Shortcut: Shift-F9). – Die Ausführung eines Makros kann man mit sog. Haltepunkte gezielt in einer bestimmten Zeile gestoppt werden (Menü "Debuggen":"Haltepunkt ein/aus"; Shortcut: F9).

Fehler beim Schreiben von Code: Teilweise erkennt der VBA-Editor einen Fehler im Code schon beim Schreiben, dann erscheint eine Meldung und die Zeile wird rot markiert. Öfter aber tritt ein Fehler erst beim Ausführen des Codes auf. Dann erscheint eine Laufzeitfehler-Meldung und die Ausführung wird an der fehlerhaften Stelle unterbrochen. Wie beim schrittweise Ausführen mit F8 ist die Zeile gelb markiert.

3 MAKROPROGRAMMIERUNG

3.1 VBA-SPRACHELEMENTE

Im Folgenden werden wichtige Sprachelement von VBA vorgestellt. Ich verwende hierfür eine bestimmte Schreibweise:

- Code wird in einer andere Schriftart dargestellt, nämlich in: `Courier New`
- Die jeweils wichtigen, bzw. besprochenen Schlüsselwörter und Elemente der Syntax setzte ich **fett**.
- Bezeichner oder Namen, die der Programmierer selbst festlegen muss, werden *kursiv* geschrieben.
- Zahlen oder Text in "Anführungszeichen" sind feste Werte. Man nennt dies auch Literale.
- Syntaxelemente, die man verwenden kann aber nicht muss (optionale Elemente), werden in eckigen Klammern **[**...**]** geschrieben: Die Klammern sind nicht Teil der Syntax und müssen weggelassen werden. Runde Klammern sind hingegen immer Teil der Syntax.
- In manchen Fällen muss zwischen alternativen Schlüsselwörtern gewählt werden. Diese werde ich mit einer Pipe **|** trennen. Dies betrifft meist optionale Elemente.
- Drei Punkte markieren, dass Platz für Anweisungen oder weitere Ausdrücke ist.
- In einzelnen Fällen werde ich einzelne Wörter als Platzhalter für bestimmte Ausdrücke oder Konstruktionen verwenden.

Grundlegende Syntax

- Eine Anweisung pro Zeile (Zeilenende ist das Anweisungsende)
- Der **Doppelpunkt :** ist das Anweisungstrennzeichen. Er erlaubt es zwei Anweisungen innerhalb einer Zeile zu schreiben.
- Der **Unterstrich _** ist das Fortsetzungszeichen. Er erlaubt es eine Anweisung über zwei Zeilen zu schreiben.
- Gross-/Kleinschreibung ist nicht signifikant
- Das **Hochkomma '** ist das Kommentarzeichen. Alles in einer Zeile nach dem Hochkomma wird nicht verarbeitet.

Namenskonventionen:

- Ein Name kann bis zu 255 Zeichen lang sein

- Das erste Zeichen muß ein Buchstabe sein
- Ansonsten sind Buchstaben, Ziffern und Unterstriche erlaubt
- Schlüsselworte sind zu meiden (die "Function sub" wäre nicht erlaubt)
- Groß-/Kleinschreibung wird nicht unterschieden

Kontext-Hilfe:

Mit F1 wird im VBA-Editor die Hilfe angezeigt. Befindet sich die Einfügemarke in einer Anweisung, dann wird direkt die Hilfsseite für diese Anweisung angezeigt (Kontexthilfe).

3.1.1 PROZEDUREN: SUB UND FUNCTION

VBA kennt drei Typen von Prozeduren: Sub, Function und Property. Letztere werden wir hier nicht behandeln, da sie nur beim Schreiben von eigenen Klassen wirklich eine Rolle spielen.
Anders als die Sub-Prozedur, die aufgerufen eine Reihe von Anweisungen ausführt, kann eine Function-Prozedur einen Wert zurückgeben. Eine Sub-Prozedur werde ich Methode und eine Function-Prozedur eine Funktion nennen.
Beide können mit oder ohne Argumente definiert werden. Argumente sind Werte, die an eine Prozedur übergeben werden; Argumente werden einer Prozedur als Variablen übergeben.
Die Syntax einer Methode ist:

```
[Private|Public]Sub Name_der_Prozedur([argliste])
   ...
End Sub
```

Die Syntax einer Funktion ist:

```
[Private|Public]Function Name ([argliste]) [as Datentyp]
   ...
   [Name = rückgabewert]
End Sub
```

Bei einer Funktion bestimmt [as datentyp] den Datentyp, den die Funktion zurückgibt. Wird nichts angegeben, wird der Datentyp implizit bestimmt (zu den Datentypen vgl. unten). Zurückgegeben wird der Wert in einer Funktion, indem der Funktion der gewünschte Wert zugewiesen wird (innerhalb der Funktion fungiert der Funktionsname wie eine Variable für den Rückgabewert).
[Private|Public] Mit diesen Schlüsselwörtern steuert man die Sichtbarkeit der Prozedur.

- Public in allen Prozeduren aller Module verwendbar
- Private nur in Prozeduren des eigenen Moduls verwendbar
- In Modulen definierte Prozeduren sind ohne Schlüsselwort Public.

Die Syntax von `argliste` ist:

```
[optional][ByVal|ByRef][ParamArray]Name_des_Arguments [as Datentyp][,...]
```

Die Elemente

- [as Datentyp] bestimmt den Datentyp des übergebenen Arguments. Wird der Datentyp nicht angegeben, ist das Argument ein Variant.
- [,...]: Mehrere Argumente können mit einem Komma getrennt angegeben werden.
- [optional]: Wird vor einem Argument das Schlüsselwort optional verwendet, muss das Argument beim Aufruf nicht angegeben werden. Mit IsMissing(Name_des_Arguments) kann man überprüfen, ob ein optionales Argument vorhanden ist (nur für Argumente vom Typ Variant).
- [ByVal|ByRef]: Bei Argumenten unterscheidet VBA, ob nur der Wert der Variable oder ein Verweis auf diese Variable (genaugenommen auf den Speicher der Variable) übergeben wird. Wird nichts angegeben, dann übergibt VBA standardmässig als Verweis. Dies ist wichtig: ändert eine Prozedur den Wert einer als Verweis übergebenen Variable, dann betrifft diese Änderung auch die Variable in der aufrufenden Prozedur. Anders gesagt: ByVal übergibt quasi die Variable selbst, ByRef nur den Wert.
- [ParamArray]: erlaubt es einer Prozedur eine beliebige Zahl von Argumenten zu übergeben. Ein als ParamArray qualifitiertes Argument ist ein Datenfeld (ein Array) (zu Datenfelder vgl. unten). Nur das letzte Argument kann ein ParaArray sein. Das Schlüsselwort kann nicht mit ByVal, ByRef oder Optional kombiniert werden.

AUFRUF VON PROZEDUREN

Eine Methode oder eine Funktion kann von anderen Prozeduren aufgerufen werden. Prozeduren können sich auch selbst aufrufen (Rekursion).

Es gibt verschiedene Varianten der Aufrufsyntax:

Prozeduren mit der Call-Methode, die Argumente müssen in Klammern gesetzt werden

```
Call Name_der_Prozedur ([arg1 [, arg2]])
```

Prozeduren ohne Call-Methode, die Argumente werden ohne Klammern geschrieben

```
Name_der_Prozedur [arg1 [, arg2]]
```

Optionale Argumente können weggelassen werden, dennoch muss an entsprechender Stelle ein Komma stehen: (arg1, , arg3).

Benannte Argumente: Anstatt die Argumente in der richtigen Reihenfolge anzugeben, kann man die Argumente auch explizit (mit Namen) übergeben:

```
Name_der_Prozedur arg2:=wert, arg1:=wert
```

Funktion als Wertzuweisung:

```
... = Name_der_Funktion ([arg1 [, arg2]])
```

Man kann den Funktionsaufruf an jeder Stelle einsetzen, an der ein entsprechender Wert erwartet wird. So kann man den Funktionsaufruf etwa als Argument eines anderen Prozeduraufrufs verwenden. Verschachteln von Funktionen:

```
[Call|...=]     Name_einer_Prozedur(Name_der_Funktion(arg1,
arg2))
```

Um eine Prozedur eines anderen Moduls aufzurufen, schreibt man vor dem Prozedurnamen den Namen des Moduls getrennt durch einen Punkt. Dies ist in VBA nicht zwingend, es sei denn, das Modul enthält eine eigene Prozedur mit demselben Namen:

```
[Call|...=] Name_des_Moduls.Name_der_Prozedur ...
```

Vorzeitiges Verlassen einer Prozedur: Die Ausführung einer Prozedur endet wenn die `End`-Anweisung erreicht ist. Mit der Anweisung **`Exit Sub`** oder **`Exit Function`** kann man eine Prozedur vorzeitig verlassen.

VERWENDEN VON METHODEN UND FUNKTIONEN IN EXCEL

Methoden und Funktionen, die in Modulen geschrieben werden, können in Excel direkt eingesetzt werden:

Methoden als Makros: Methoden ohne Argumente können als Makros aufgerufen werden.

```
Sub MethodeOhneArgumente()
    MsgBox "test"
End Sub
```

Verwenden können sie Makros:

- über den Makrodialog
- mittels einer Tastenkombination (Optionen im Makrodialog)
- über einen Menüpunkt auf dem Menüband (Anpassen durch Rechtsklick auf Menüband)
- durch Anklicken eines Objekt auf einem Arbeitsblatt. Hierzu weisen Sie einem Diagramm, einem grafischen Element oder einer Textbox über das Kontextmenü ein Makro zu.

Besonders geeignet um Makros direkt von einem Arbeitsblatt zu starten, ist die "Schaltfläche" der "Formularsteuerelemente". Die Formularsteuerelemente finden Sie auf dem Reiter Entwicklungstools unter "Einfügen". Achtung: nicht die "Active-X Steuerelemente".

Methoden mit Argumenten kann man nicht direkt in Excel verwenden, sondern nur aus anderen Prozeduren:

```
Sub MethodeMitArgument(arg1)
    MsgBox arg1
End Sub
```

Funktionen als "benutzerdefinierte Funktionen": Jede in VBA geschriebene Funktion können Sie wie die eingebauten Funktionen in einem Tabellenblatt verwenden. Im Funktionsassistenten werden sie unter der Kategorie "Benutzerdefiniert" angezeigt. Argumente können wie bei normalen Funktionen auch Zellbezüge sein.
Wenn die Funktion nicht in einem Modul der Arbeitsmappe selbst (oder in einem AddIn) definiert ist, müssen Sie vor dem Funktionsnamen den Namen der Mappe (den Speicherort) angeben. Dies gilt auch für die "Persönliche Arbeitsmappe" (meist "PERSONAL.XLSB"). Ein Beispiel: =PERSONAL.XLSB!FunktionMitArgument(B2)

```
Function FunktionMitArgument(arg1)
    FktMitArg = arg1
End Function

Function FunktionOhneArgument()
    FktOhneArg = "test"
End Function
```

Der Rückgabewert der Funktion ist dann der Wert der Zelle. Tritt in der Funktion ein Fehler auf, dann erscheint in der Zelle der Fehlerwert "#Wert!". Oft tritt in benutzerdefinierten Funktionen ein Fehler auf, weil falsche Argumente angegeben werden. Wird hingegen keine entsprechende benutzerdefinierte Funktion gefunden, dann ist der Wert der Zelle der Fehlerwert "#Name?".
Benutzerdefinierte Funktionen unterliegen in ihrer Verwendung keiner mir bekannten Restriktion. Sie können auf sie verweisen, sie in andere Funktionen verschachteln, andere Funktionen in sie verschachteln usw. Wenn Sie sie massenhaft verwenden, werden Sie aber merken, dass benutzerdefinierte Funktionen etwas langsamer sind als die eingebauten.

3.1.2 DATENTYPEN: VARIABLEN UND KONSTANTEN

Variablen und Konstanten sind gewissermaßen Behälter für einen Wert. Sie bestehen aus einem Namen oder Bezeichner, dem man einen Wert zuweist. Konstanten haben einen festen Wert, der Wert einer Variablen kann verändert werden. Für beide kann man festlegen, welche Art von Werten ihnen zugewiesen werden können; man legt den Datentyp der Variablen fest. Genaugenommen ist eine Variablen oder eine Konstante eine benannte Speicherposition, der Datentyp legt dann fest, wie viel Speicher reserviert wird. Für kleinere Programme, wie Mak-

ros, ist das weniger wichtig. Dennoch ist es empfehlenswert, den Datentyp zu bestimmen, um Fehler zu verhindern.

Um eine Variable oder eine Konstante verwenden zu können, muss man sie in VBA nicht zwingend explizit deklarieren. Es ist möglich, einem Variabelenamen direkt einen Wert zuzuweisen (implizite Deklaration).

Explizite Deklaration einer Variablen:

```
Dim Name_der_Variablen [as Datentyp]
```

Man kann in einer Zeile auch mehrere Variablen definieren:

```
Dim Name_der_Variablen [as Datentyp], Name_der_Variablen2 [as Datentyp]
```

Implizite Deklaration: VBA erlaubt eine implizite Deklaration von Variablen, indem man einfach einem Namen einen Wert zuweist.

```
Name_der_Variablen = wert
```

Deklaration einer Konstanten:

```
Const Name_der_Konstanten [as Datentyp] = wert
```

Bei der Deklaration einer Variablen wird sie gleichzeitig initialisiert, d.h. sie erhält einen Standard-Wert (Null-Wert): Bei numerischen Datentypen ist das 0, bei einem String die leere Zeichenkette: "". Bei einem Boolean ist der Null-Wert `false` und bei einem Variant `empty`. Wird bei der Deklaration von Variablen kein Datentyp angegeben, dann definiert VBA diese als `Variant`.

Datentyp	Größe	Wertbereich	Kürzel
Boolean	2 Bytes	True(-1) oder False(0)	bln
Byte	1 Byte	Ganze Zahlen von 0 bis 255	byt
Integer	2 Bytes	Ganze Zahlen von -32768 bis +32767	int
Long	4 Bytes	Ganze Zahlen von -2147483648 bis +2147483647	lng
Single	4 Bytes	±3,402823e38...±1,401298e-45	sng
Double	8 Bytes	±4,94065645841247e308 bis ±4,94065645841247e-324	dbl
Currency	8 Bytes	-922337203685477,5808 bis 933337203685477,5807	cur
Date	8 Bytes	Datumsangaben	dat
String	10 Bytes	Bis zu 2 Milliarden beliebige ASCII-Zeichen	str
Variant	22 Bytes	Beliebiger Wert oder Objekt	var

Zuweisung: Die Wertzuweisung erfolgt mit dem Gleichheitszeichen (=):

```
Name_der_Variable = wert
```

Der zugewiesene Wert kann ein fester Wert oder der Rückgabewert einer Funktion (oder Objekt-Eigenschaft) sein.

Für feste Werte gilt folgende Syntax:

- Text muss in doppelten Anführungszeichen stehen.
- Zahlen werden ohne Anführungszeichen geschrieben. Dezimaltrennzeichen ist der Punkt. Mit "E+" oder "E-" kann man Zahlen in einer Exponentialschreibweise angeben.
- Ein Datum wird in englischer Schreibweise zwischen zwei Rauten # angegeben (#Monat/Tag/Jahr Stunde**:**Minute**:**Sekunde AM|PM#); etwa: #3/28/2013 10:15:30 AM#)

Vordefinierte Konstanten: In VB und VBA sind Konstanten für viele Eigenschaftswerte definiert. Sie erleichtern das Schreiben und das Lesen von Code. Hinweise gibt es in der Hilfe oder oft im Tooltip, bzw. der Quickinfo. Besonders hilfreich finde ich die Zeichenkonstanten `vbCr` für einen Absatz und `vbTab` für einen Tabulator.

Tipp: Mit der Anweisung `Option Explicit` am Anfang eines Moduls erzwingt man eine explizite Deklaration von Variablen. Dies verhindert, dass man durch einen Tippfehler eine neue Variable kreiert, anstatt eine vorhandene zu verwenden.

OBJEKTVARIABLEN

In der objektorientierten Programmierung kann eine Variable anstatt für einen Wert auch für ein Objekt stehen. Die Deklaration erfolgt ähnlich wie bei einer normalen Variablen mit "as" gefolgt vom Namen der Klasse des Objekts. Ähnlich wie der Datentyp `Variant` gibt es einen Datentyp für unbestimmte Objekte: `Object`.

Deklaration:

```
Dim Name_der_Variablen as Object
Dim Name_der_Variablen as klassenname
```

Zuweisung:

```
Set Name_der_Variablen = instanz
```

Zuweisung und Initialisierung einer neuen Instanz (selten in der Makro-Programmierung):

```
Set Name_der_Variablen = New klassenname
```

Aufheben der Zuweisung

```
Set Name_der_Variablen = Nothing
```

Nothing ist der Null-Wert einer Objektvariablen; er wird ihr bei der Initialisierung zugewiesen.

Ein Beispiel mit der Klasse `Range` und der Instanz der Zelle C4

```
Dim Eine_Zelle As Range
Set Eine_Zelle = Range("C4")
```

Zugreifen auf eine Eigenschaft oder Methode einer Objektvariablen:

Mit dem Punkt greift man auf Eigenschaften oder Methoden von Objekten zu. Eine Methode oder Eigenschaft kann zusätzlich Argumente verlangen.

Syntax:

```
Objekt_Variable.Name_der_Eigenschaft[(argumente)]
... = Objekt_Variable.Name_der_Eigenschaft[(argumente)]
Objekt_Variable.Name_der_Eigenschaft[(argumente)] = ...
```

Ist eine Objekteigenschaft selbst ein Objekt, dann kann man mit einem weiteren Punkt auf deren Methoden und Eigenschaften zugreifen.

With-Anweisung: Mit der With-Anweisung können Sie eine Reihe von Anweisungen für ein Objekt bündeln, ohne dieses mehrfach angeben zu müssen:

```
With Objekt_Variable
   .Eigenschaft1 = wert1
   .Eigenschaft2 = wert2
   ... = .Eigenschaft2
   .Methode[(argumente)]
End With
```

GÜLTIGKEITSBEREICH UND LEBENSDAUER

Gültigkeitsbereich: Eine Variable (oder eine Konstante) ist nur in dem Bereich gültig, in welchem sie definiert wurde. Eine Variable kann auf der Ebene einer Prozedur oder eines Moduls definiert werden. Wird eine Variable zwischen `Sub` und `End Sub` (bzw. `Function`) definiert, dann kann sie nur in dieser Prozedur verwendet werden. Um eine Variable für ein ganzes Modul zu definieren, muss die Variablen-Deklaration vor der ersten Prozedur geschrieben werden.

Mit dem Schlüsselwort `Public` kann eine auf Modulebene definierte Variable auch in anderen Modulen verwenden werden. Man nennt diese dann eine global gültige Variable.

Im folgenden Beispiel werden vier Variablen definiert:

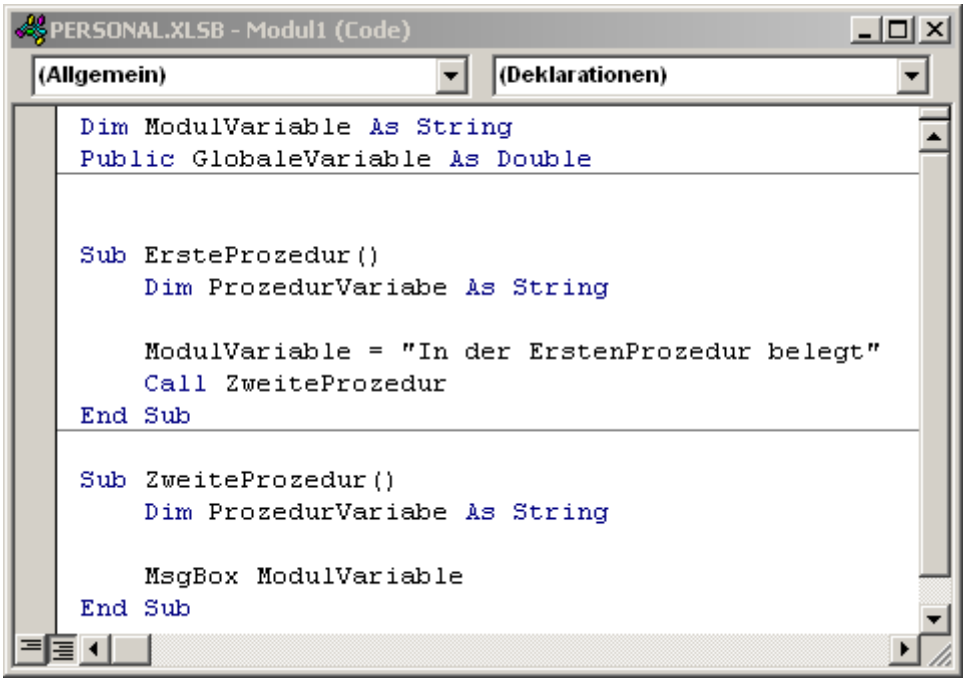


Abbildung 10: Deklaration von Variablen

Die Variable "ModulVariable" ist nur in diesem Modul "Modul1" gültig. Die Variable "GlobaleVariable" könnte auch in anderen Modulen verwendet werden (Syntax: `Modul1.GlobaleVariable`). Die Variable "ProzedurVariable" wird auf Prozedurebene definiert. In diesem Beispiel gibt es in den zwei Prozeduren je eine Variable mit dem gleichen Namen. Sie sind, da nur auf Prozedurebene gültig, unabhängig voneinander.

Innerhalb eines Gültigkeitsbereiches muss der Variablennamen eindeutig sein. Wird auf Prozedurebene eine Variable mit dem gleichen Namen wie auf Modulebene definiert, dann wird letztere für diese Prozedur überlagert.

Lebensdauer: Eine Variable behält ihren zugewiesenen Wert solange wie die Programmausführung den Gültigkeitsbereich nicht verlässt. Wird, wie in unserem Beispiel, aus der "ErstenProzedur" die "ZweiteProzedur" aufgerufen, dann behält eine in der "ErstenProzedur" definierte Variable ihren zugewiesenen Wert. Die Ausführung der "ErstenProzedur" wird durch den Aufruf einer anderen Prozedur nicht beendet (erst `End Sub` beendet die Ausführung). Bei einem wiederholten Aufruf von "ZweiteProzedur" werden ihre Variablen wieder neu initialisiert. Auf Modulebene definierte Variablen leben solange, wie eine Prozedur des Moduls noch ausgeführt wird.

Weiterführender Hinweis: Mit dem Schlüsselwort `Static` kann man die Lebensdauer von Variablen verändern.

Am Rande: Der Befehl `MsgBox` zeigt eine Meldung in einem Dialogfeld an. In diesem Beispiel wird einfach der Wert der Variable "ModulVariable" angezeigt.

Tipp: Es ist gute Programmierpraxis Gültigkeitsbereich und Datentyp einer Variable durch ein Präfix im Variablennamen zu markieren. Hierfür eigenen sich die Kürzel aus obiger Datentypen-Tabelle und ein "m" für modul-spezifische und "g" für globale Variablen. Bsp.: "**str***Name*" oder "**gDbl***Name*".

KONVERTIERUNGSFUNKTIONEN

Es gibt in VBA spezielle Funktionen, um den Wert einer Variablen einer anderen Variablen mit einem anderen Datentyp zuzuweisen. In vielen Fällen funktioniert diese Konvertierung auch implizit.

- CBool (expression) → Boolean
- CByte (expression) → Byte
- CCur (expression) → Currency
- CDate (expression) → Date
- CDbl (expression) → Double
- CDec (expression) → Decimal
- CInt (expression) → Integer
- CLng (expression) → Long
- CSng (expression) → Single
- CVar (expression) → Variant
- CStr (expression) → String

Spezialfälle

- Val (string-expression) — Eine Zahl am Anfang einer Zeichenkette wird extrahiert. Bsp. "5 Stunden" gäbe den numerischen Wert 5 zurück.
- Str (number-expression) — Die Zahl wird so als Zeichenkette dargestellt, dass die Funktion Val den richtigen Wert liefern würde (Dezimalzeichen)

Ein kleines Beispiel

```
Dim EineZahl as Integer
Dim EinText as String
EineZahl = 5
EineZahl = EineZahl + 6
EinText = CStr(EineZahl)
                'Ginge auch implizit: EinText=EineZahl
EinText = "Resultat der Summe 5+6=" & EinText
                'Text verketten (& ist Verkettungsoperator)
Msgbox EinText
```

3.1.3 DATENTYPEN: DATENFELDER UND AUFLISTUNGEN

Ein spezieller Typ von Variablen sind Datenfelder und Kollektionen. Sie erlauben es in einer Variablen mehrere Werte zu speichern und diese über einen Index abzurufen. Während in

einem Datenfeld nur Werte eines Datentyps gespeichert werden können, kann man in einer Kollektion unterschiedliche Daten aufbewahren.

DATENFELD: ARRAY

Ein Datenfeld ist ein Satz aufeinanderfolgender, indizierte Elemente. Alle Elemente eines Datenfeldes haben den gleichen Datentyp. Ein Datenfeld kann eine oder mehrere Dimensionen haben. Ein eindimensionales Datenfeld ist ein Vektor, ein zweidimensionales eine Matrix. Beim Deklarieren eines Datenfeldes muss man für jede Dimension den größten verfügbaren Index angeben, zusätzlich kann man auch den kleinsten Index definieren.

Deklaration eines eindimensionalen Datenfeldes:

```
Dim Name_des_Arrays([untergrenze to] obergrenze) [as Datentyp]
```

Deklaration eines mehrdimensionalen Datenfeldes:

```
Dim Name_des_Arrays([ug to] og, [ug to] og [,[ug to] og, ...]) [as Datentyp]
```

Standardmäßig ist die Untergrenze Null. Dies kann verwirren, da man ein Element mehr im Datenfeld hat, als man mit der Obergrenze angibt. Deklariert man `Dim v(2)`, dann hat man einen Array mit drei Elementen (die Elemente 0, 1 und 2).

Tip: Mit der Anweisung `Option Base 1` am Modulanfang legt man fest, dass Datenfelder standardmäßig mit einer Untergrenze von 1 definiert werden.

Element eines Arrays: Ein Element eines Array wird abgerufen oder festgelegt, indem man die entsprechenden Indizes in jeder Dimension angibt. In einem zweidimensionalen nullbasierten Array wäre das erste Element: v(0, 0). Array-Elemente kann man wie andere Variablen des entsprechenden Typs verwenden.

Wertzuweisung:

```
Name_des_Arrays(m[, n ...]) = wert
```

Wertrückgabe:

```
... = Name_des_Arrays(m[, n ...])
```

Beispiel: Das Text-Datenfeld "MeinArray" hat zwei Dimensionen mit einer Obergrenze von 5 bzw. 4:

```
Dim MeinArray(5, 4) as String
MeinArray(0, 0) = "Das erste Element"
MeinArray(5, 4) = "Das letzte Element"
```

In gewissen Fällen kann es sinnvoll sein, ein Datenfeld dynamisch zu deklarieren, d.h. ohne festgelegte Größe.

Dynamische Deklaration:

```
Dim Name_des_Arrays() [as Datentyp]
```

Festlegen der Grösse (Initialisiert den Array neu, alle Werte werden gelöscht):

```
ReDim Name_des_Arrays([ug to] og [,[ug to] og, ...])
```

Mit ReDim können Variablen oder Funktionsrückgabewerte verwendet werden, um die Grenzen festzulegen.

Mit dem Schlüsselwort `Preserve` kann ein Array vergrößert oder verkleinert werden, ohne dass bestehende Elemente gelöscht werden.

```
ReDim Preserve Name_des_Arrays([ug to] og [,[ug to] og, ...])
```

Wichtige Funktionen für Datenfelder:

- **UBound**(datenfeldname[, dimension]) gibt die Obergrenze der angegebenen Dimension des Datenfeldes zurück
- **LBound**(datenfeldname[, dimension]) gibt die Untergrenze zurück
- Mit **IsArray**(Variablen_Name) kann man feststellen, ob eine Variable ein Datenfeld ist oder nicht. Die Funktion gibt wahr oder falsch zurück.
- `Split` und `Join`: Ein String in einen Array vom Typ String aufspalten und umgekehrt.

AUFLISTUNG: COLLECTION

Eine Auflistung ist eine geordnete Folge von Elementen. Die Elemente einer Auflistung können verschiedenen Typs sein. Jedem Element einer Auflistung kann ein Schlüssel zugeordnet werden, mit dessen Hilfe es wieder abgerufen werden kann. – Eine Auflistung ist eigentlich keine Datentyp, sondern ein Objekt.

Deklarationssyntax:

```
Dim Name_der_Auflistung as Collection
```

Initialisierung:

```
Set Name_der_Auflistung = New Collection
```

Deklaration mit gleichzeitiger Initialisierung:

```
Dim Name_der_Auflistung as New Collection
```

Zuweisung:

```
Set Name_der_Auflistung = EineAuflistung
```

Element zufügen:

```
Name_der_Auflistung.Add element[, schlüssel, vor, nach]
```

- *element* ist eine Variable, ein Wert oder ein Objekt, das zur Auflistung hinzugefügt werden soll

- *schlüssel* ist eine eindeutige Zeichenfolge. Sie kann anstelle des Index verwendet werden, um auf das Elemente zuzugreifen
- *vor* oder *nach*: wird für *vor* oder für *nach* ein Index oder eine Schlüssel eines Elementes der Auflistung angegeben, wird das neue Elemente davor oder danach eingereiht. Man kann nur eines der beiden angeben.

Auf Element zugreifen:

```
Name_der_Auflistung.Item(arg)
```

- *arg* ist als numerischer Wert der Index des Elements
- *arg* ist als Zeichenfolge der Schlüssel des Elements
- ".Item" kann auch weggelassen werden

Element löschen:

```
Name_der_Auflistung.Remove(arg)
```

- *arg* kann wie bei .Item der Index oder ein Schlüssel sein

Anzahl der Elemente in der Auflistung

```
Name_der_Auflistung.count
```

Auflistungen definiert man in den Anfängen der Makroprogrammierung wohl eher selten selbst. Sie werden Ihnen aber als Eigenschaften von vielen Excel-Objekten begegnen. Das Arbeitsmappen-Objekt besitzt als eine Eigenschaft eine Auflistung aller dazu gehörender Tabellenblätter.

Beispiel:

```
Dim meineAuflistung As New Collection
meineAuflistung.Add "ein Element", "ele1"
meineAuflistung.Add "ein zweites Element", "ele2"
meineAuflistung.Add 55, "ele3"

Debug.Print "Anzahl der Elemente: " & meineAuflistung.Count
Debug.Print "Element 1: " & meineAuflistung.Item(1)
Debug.Print "Element 2: " & meineAuflistung(2)
Debug.Print "Element mit dem Schlüssel ele3: " &
meineAuflistung("ele3")
```

Am Rande: Der Befehl `Debug.Print` hilft beim Ausprobieren von Code. Er schreibt den als Argument übergebenen Wert in den sog. Direktbereich (Menü: "Ansicht":"Direktfenster"). In den Argumenten verwende ich hier den Verkettungsoperator "&".

3.1.4 OPERATOREN

Operatoren werden verwendet, um in Anweisungen, meist Wertzuweisungen zu Variablen, mit Werten oder Variablen bestimmte Operationen (Rechnen, Vergleichen, Verketten) durchzuführen.

ARITHMETISCHE OPERATOREN:

Grundrechenarten	+ - * /	a + b	
Potenzierung	^	a ^ b	Wurzel a ^ (1/b)
Ganzzahldivision	\	a \ b	Division der auf natürliche Zahlen gerundeten Operanden (3.4\2=1 aber 3.5\2=2)
Modulo-Division	mod	a mod b	Rest einer ganzzahligen Division (3.4 mod 2=1 aber 3.5 mod 2=0)
Negation	-	-a	

LOGISCHE OPERATOREN:

			a = falsch b = falsch	a = wahr b = falsch	a = falsch b = wahr	a = wahr b = wahr
Konjunktion	and	a and b	FALSCH	FALSCH	FALSCH	WAHR
Äquivalenz	eqv	a eqv b	WAHR	FALSCH	FALSCH	WAHR
Implikation	imp	a imp b	WAHR	FALSCH	WAHR	WAHR
Negation	not	not a				
Disjunktion	or	a or b	FALSCH	WAHR	WAHR	WAHR
Antivalenz	xor	a xor b	FALSCH	WAHR	WAHR	FALSCH

VERGLEICHSOPERATOREN

Vergleich	< <= >= >	a < b	
Gleichheit	=	a = b	
Ungleichheit	<>	a <> b	
Referenzvergleich	is	a is b	Vergleich von Objekten
Ähnlichkeit	like	a like muster	Platzhalter für Muster: ? und *

Hinweis: Für Vergleiche gilt [0..9]<[A..Z]<[a..z] (entspricht der ASCII-Codierung)

VERKETTUNGSOPERATOREN

Verkettung	&	a & b	a und b werden als String interpretiert ("2" + 2 = "2" + "2" = "22")
Verkettung	+	a + b	a und b müssen vom Typ String sein. VBA versucht die Operanden als Zahlen zu interpretieren: "2" & "2" = "22", aber "2" + 2 = 4

3.1.5 KONTROLLFLUSS: VERZWEIGUNGEN UND SCHLEIFEN

Ein zentrales Element der Programmierung ist der Kontrollfluss. Der Kontrollfluss bestimmt den Ablauf eines Programms. Zum einen gibt es Verzweigungen, die abhängig von Bedingungen steuern, welche Teile eines Programmes ausgeführt werden. Zum anderen gibt es Schlaufen, mit denen man einen Teil des Programmes wiederholen kann bis eine Bedingung erfüllt ist.

Eine Bedingung ist meist ein Ausdruck, der wahr oder falsch ist. Als Bedingung kann eine Variable vom Typ Boolean, ein Vergleich oder eine Funktion (Typ Boolean) verwendet werden. Mittels logischer Operatoren können komplexe Bedingungen formuliert werden.

IF-VERZWEIGUNG

Die If-Anweisung prüft eine oder mehrere Bedingungen und setzt die Ausführung des Codes davon abhängig an entsprechender Stelle fort. Syntax:

```
If Bedingung Then
  [Anweisungen]
[ElseIf ElseIfBedingung Then
  [elseIfAnweisungen]]
[ElseIf ElseIfBedingung Then
  [elseIfAnweisungen]]
[Else
  [elseAnweisungen]]
End If
```

In einer Zeile:

```
If Bedingung Then [Anweisungen] [Else elseAnweisungen]
```

Bei der Ausführung wird geprüft, ob die "If"-Bedingung erfüllt ist (wahr ist). Ist dies der Fall, dann wird die Ausführung im If-Block mit den *Anweisungen* fortgesetzt. Ist die Bedingung nicht erfüllt, wird geprüft, ob (falls vorhanden) die ElseIfBedingung erfüllt ist. Eine If-Verzweigung kann mehrere ElseIF-Blöcke enthalten. Wenn die IF-Bedingung und keine ElseIf-Bedingung erfüllt ist, wird, falls vorhanden, die elseAnweisungen im Else-Block ausgeführt. Dem If, ElseIf oder Else müssen nicht zwingend Anweisungen folgen.
If-Anweisungen können beliebig verschachtelt werden. Jeder Anweisungs-Teil kann also wieder if-Verzweigungen enthalten.

Beispiel:

```
Dim intWert1 As Integer, intWert2 As Integer
intWert1 = 5
intWert2 = 10
If intWert1 > intWert2 Then
    MsgBox "grösser"
ElseIf intWert1 < intWert2 Then
    MsgBox "kleiner"
    If intWert1 < 0 Then
        MsgBox "unter null"
    End If
Else
    MsgBox "gleich"
End If
```

Tipp: Eine ähnliche Funktionsweise wie eine If-Verzeigung haben die Funktionen `IIf()` und `Switch()`.

SELECT CASE-VERZWEIGUNG

Die Select Case-Anweisung prüft einen beliebigen numerischen Ausdruck oder einer Zeichenfolge darauf, ob er mit einer Reihe von Fällen übereinstimmt. Syntax:

```
Select Case Testausdruck
  Case Ausdrucksliste
    [Anweisungen]
  [Case Ausdrucksliste2
    [Anweisungen2]]
  [...]
  [Case Else
    [elseAnweisung]]
End Select
```

Die Ausdruckliste kann folgende Formen haben:

- einen einzelnen Wert: `5`
- einen durch das Schlüsselwort **to** definierten Bereich: `4 to 8`
- einen durch **Is** und Vergleichsoperator eingeleiteten Vergleich: `Is < 20`
- eine durch Komma getrennte Kombination: `1, 3 to 6, Is > 20`

Die zu prüfenden Ausdrücke müssen vom selben Datentyp sein wie der Testausdruck.

Eine Select Case-Anweisung kann mehrere Case-Abschnitte enthalten. Ein Case-Else-Block ist nicht zwingend.

Die Case-Anweisung arbeitet die Ausdrucklisten ab, bis es eine Übereinstimmung gefunden wird. Das Programm führt die Anweisungen in diesem Block aus und springt dann zur `End Select`-Zeile.

Beispiel:

```
Dim intTestWert As Integer
intTestWert = 11
Select Case intTestWert
    Case 1
        MsgBox "1"
    Case 3, 5, 6, 11
        MsgBox "3, 5, 6, 11"
    Case 10 To 100
        MsgBox "gross"
    Case Is > 100
        MsgBox "sehr gross"
    Case Else
        MsgBox "irgend was anderes"
End Select
```

Tipp: Eine ähnliche Funktionsweise hat die Funktion `Choose()`.

DO ... LOOP-SCHLEIFE

Eine Do..Loop Schleife wiederholt eine Reihe von Anweisungen bis eine (sofern vorhanden) Until-Bedingung erfüllt oder eine While-Bedingung nicht mehr erfüllt ist. Eine While- oder

Until-Bedingung muss nicht vorhanden sein. Eine Do...Loop-Schleife kann auch jederzeit mit einer `Exit Do` Anweisung verlassen werden. Diese steht typischerweise in einer If-Verzweigung. Syntax:

```
Do [While|Until bedingung]
   [Anweisungen]
   [[If bedingung then] Exit Do]
   [Anweisungen]
Loop
```

oder:

```
Do
   [Anweisungen]
   [[If bedingung then] Exit Do]
   [Anweisungen]
Loop [While|Until bedingung]
```

While und Until unterscheiden sich darin, dass mit While die Schleife solange wiederholt wird, wie die Bedingung wahr ist, während ein Schleife mit Until beendet wird, wenn die Bedingung wahr wird. – Ob die Bedingung beim Do oder beim Loop steht, entscheidet darüber, ob erst die Bedingung geprüft wird oder ob erst die Anweisungen ausgeführt werden.

Eine Alternative zu Do...Loop ist die `While...Wend` Anweisung. Sie entspricht genau der Do While...Loop Schleife.

Achtung: Schleifen bergen das Risiko, dass die Abbruchbedingung nie erfüllt wird und somit die Schleife endlos ausgeführt wird. Das Programm hat sich, wie man so schön sagt, aufgehängt.

Beispiel:

```
Sub EineEndlosSchleife()
    Dim ii As Long
    Do Until ii < 0
        ii = ii + 1
        Selection = ii
    Loop
End Sub
```

In Excel können Sie die Ausführung eines Makros immer mit Strg-Break/Pause unterbrechen.

FOR … NEXT-SCHLEIFE

Mit einer For..Next-Schleife wird eine Reihe von Anweisungen eine festgelegte Anzahl mal wiederholt. Syntax:

```
For Zähler = Anfang To Ende [Step Schritt]
   [Anweisungen]
   [[If Bedingung then] Exit For]
   [Anweisungen]
Next [Zähler]
```

Der `Zähler` ist eine Variable, die vom Wert `Anfang` bei jedem Durchlauf um 1 oder den als `Schritt` angegebene Wert erhöht wird. Wenn `Zähler` grösser als der `Ende`-Wert ist, wird die Schleife verlassen. Mit `Exit For` kann die Schleife jederzeit beendet werden.

`Zähler`, `Anfang`, `Ende` und `Schritt` sind numerische Werte; sie müssen nicht ganzzahlig sein. Der `Zähler` muss eine Variable sein und es empfiehlt sich, ihn als einen `Integer` zu deklarieren. `Anfang`, `Ende` und `Schritt` können feste Werte, Variablen oder der Rückgabewert eines Funktionsausrufs sein.

Beispiel:

```
Dim ii As Integer
For ii = 2 To 20 Step 2
    Debug.Print ii
Next ii
```

Typische Anwendung: Die häufigste Anwendung einer For...Next Schleife ist der Durchlauf durch die Elemente eines Datenfeldes:

```
' Array deklarieren:
Dim strFeld(3) As String
' Array füllen:
strFeld(0) = "Alle"
strFeld(1) = "Elemente"
strFeld(2) = "eines"
strFeld(3) = "Arrays"
' Array mit For..Next auslesen:
Dim intIndex As Integer
For intIndex = LBound(strFeld) To UBound(strFeld)
    Debug.Print strFeld(intIndex)
Next intIndex
```

FOR EACH ... NEXT-SCHLEIFE

Eine ForEach-Schleife wiederholt eine Reihe von Anweisungen für jedes Element einer Auflistung oder eines Datenfeldes. Syntax:

```
For Each Element In Gruppe
   [Anweisungen]
   [[If Bedingung then] Exit For]
   [Anweisungen]
Next [Element]
```

Als `Gruppe` muss der Name des Datenfeldes oder der Auflistung stehen. `Element` muss eine Variable des Typs `Variant` oder eine Objektvariable sein. Letzteres verwendet man

dann, wenn man eine Auflistung von Objekten einer bestimmten Klasse durchlaufen will, etwa die Tabellen einer Arbeitsmappe.

Typische Anwendung: ForEeach-Schleifen werden meist verwendet, wenn ein Objekt, bspw. eine Arbeitsmappe, eine Auflistung anderer Objekte, bspw. Tabellenblätter, enthält. Die aktuelle Arbeitsmappe (Instanz: `ActiveWorkbook`) besitzt als Eigenschaft die Auflistung `Sheets`, die alle Tabellen (Klasse: `Worksheet`) enthält. Eine der Eigenschaften des `Worksheet`-Objekts ist der Name der Tabelle (`Name`):

```
Dim wshEle As Worksheet
For Each wshEle In ActiveWorkbook.Sheets
    Debug.Print wshEle.Name
Next
```

3.1.6 KONTROLLFLUSS: GOTO UND SPRUNGMARKEN

Mit der Anweisung `GoTo` kann man innerhalb einer Prozedur zu einer mit einer Sprungmarke markierten Zeile springen. Eine Sprungmarke ist entweder eine Zahl (eine Zeilennummer) oder ein Ausdruck, der mit einem Doppelpunkt endet. Die Ausführung des Codes wird nach der Zeile mit der Sprungmarke fortgesetzt. Eine GoTo-Anweisung steht typischerweise in einer If-Verzweigung.

Syntax:

```
GoTo Zeile
...
Zeile[:]
...
```

Zuviele GoTo-Anweisungen machen einen Code unlesbar und sollten daher vermieden werden.

Tipp: GoSub ist eine Erweiterung der GoTo-Anweisung.

3.1.7 KONTROLLFLUSS: FEHLERBEHANDLUNG

Bei der Ausführung eines Programmes können Fehler auftreten. Viele Fehler kann man durch einen gut programmierten Kontrollfluss vermeiden. Man prüft etwa mit einer If-Verzweigung, ob eine Aktion überhaupt erlaubt ist. Es ist aber kaum möglich, beim Programmieren alle Eventualitäten zu bedenken oder gar zu berücksichtigen. Die Methoden zur Fehlerbehandlung erlauben es Unvorhergesehenes aufzufangen, ohne dass die Ausführung abgebrochen werden muss. Der größte Fehler beim Programmieren ist das Fehlen einer Fehlerbehandlung.

Eine Fehlerbehandlung wird mit der `On Error` Anweisung aktiviert. Es gibt zwei Varianten:

`On Error Resume Next`

- Tritt in einer Zeile ein Fehler auf, wird die Ausführung des Code in der nächsten Zeile fortgesetzt.

Oder:

`On Error GoTo` *`Zeile`*

- Tritt ein Fehler auf, wird die Ausführung des Codes nach der Sprungmarke fortgesetzt.
- Die Sprungmarke für die Fehlerbehandlung steht meist am Ende der Prozedur nach einer `Exit Sub|Function` Anweisung. Eine Sprungmarke ist entweder eine Zahl (eine Zeilennummer) oder ein Ausdruck, der mit einem Doppelpunkt endet.
- Mit der `Resume-`Anweisung kann man eine Fehlerbehandlung wieder verlassen:
  - `Resume`: Die Ausführung des Codes wird mit der den Fehler auslösenden Anweisung fortgesetzt. Die Anweisung wird wiederholt (Achtung: Endlosschleife!).
  - `Resume Next`: Die Ausführung wird in der Zeile nach dem Fehler fortgesetzt.
  - `Resume` *`Zeile`*: Die Ausführung wird nach der Sprungmarke `Zeile` fortgesetzt.

Informationen zu einem aufgetretenen Fehler werden im `Err`-Objekt gespeichert. Wichtig sind die beiden Eigenschaften `.Number` und `.Description`. Die Description-Eigenschaft enthält eine Beschreibung des Fehlers. Die Number-Eigenschaft ist ein Zahlenwert, der hilft den Fehler für eine Behandlung zu identifizieren (bspw. mittels einer If- oder besser einer Case-Verzweigung). Number ist die Standardeigenschaft des Err-Objekts und kann weggelassen werden. Die Methode `.clear` setzt das Err-Objekt zurück.

Mit der Methode `.raise(Nummer, Quelle, Beschreibung)` kann man einen Fehler auch selbst auslösen. Dies bietet sich an, um in einer Fehlerroutine einen nicht vorgesehenen Fehler weiterzureichen an die aufrufende Prozedur: `Err.Raise Err`

Ist kein Fehler aufgetreten, dann gilt `Err=0`. Das Err-Objekt wird mit der Clear-Methode, der Resume-Anweisung oder durch Beenden der Prozedur (eine Exit oder End Sub|Function Anweisung wurde erreicht) zurückgesetzt.

Fehlerbehandlung deaktivieren und Err-Objekt zurücksetzen:

```
On Error Goto 0
```

Anwendungsmuster für `On Error Goto` zeile:

```
Sub Beispiel()
  [Anweisung]
  On Error Goto FehlerBehandlung
  [Anweisung]
ProzedurAbschliessenSprungmarke:
  [Anweisung]
Exit Sub
FehlerBehandlung:
  [Select Case Err
    Case fehlernummer1
      [Fehlerbehandlung]
      Resume Next
    Case fehlernummer2
      [Fehlerbehandlung]
      Resume
    Case Else
      Msgbox Err.Description, vbCritical _
                  , "Fehler " & Err.Number
      Resume ProzedurAbschliessenSprungmarke
  End Select]
End Sub
```

Anwendungsmuster für `On Error Resume Next`:

```
[Anweisung]
On Error Resume Next
[Anweisung]
[If Err then
  [Fehlerbehandlung]
  [Err.Clear]
end if]
[Anweisungen]
```

3.1.8 Wichtige Schlüsselwörter, Anweisungen und Funktionen

Im Folgenden werde ich Funktionen, Methoden und Objekte thematisch geordnet auflisten. Dies soll Ihnen einen Anhaltspunkt geben, um die Befehle in der VBA-Hilfe nachschlagen zu können. Sie können die Befehle entweder im Suchfeld eingeben oder über die Kontexthilfe aufrufen (Cursor in den Befehl im Editor platzieren und mit F1 die Hilfe aufrufen). Mit ein bisschen Englischkenntnissen erschließt sich die Funktion meist von selbst.

Zeichenfolgen-Schlüsselwörter	
Teile von Zeichenfolgen	Mid, Left, Right
Position eines Zeichen in einer Zeichenfolge	InStr, InStrRev
Beschneiden einer Zeichenfolge	Trim, LTrim, RTrim
Bestimmen der Länge einer Zeichenfolge	Len

Erstellen einer Zeichenfolge mit sich wiederholenden Zeichen	Space, String
Formatieren einer Zeichenfolge	Format
Umwandeln in Groß- oder Kleinschreibung	LCase, UCase
Umwandeln von Zeichenfolgen	StrConv
Vergleichen zweier Zeichenfolgen	StrComp
Zeichenfolge zu Array	Join, Split
Ersetzen eines Teils einer Zeichenfolge	Replace
Zeichenkonstanten	vbTab, vbCr, vbLf, vbCrLf
Arbeiten mit ASCII- und ANSI-Werten. Etwa zum Schreiben von Anführungszeichen: Chr(34)	Asc, Chr
Ausrichten einer Zeichenfolge	LSet, RSet
Festlegen von Vergleichsregeln für Zeichenfolgen	Option Compare

Datum- und Uhrzeit-Schlüsselwörter	
Abrufen des aktuellen Datums oder der Zeit	Date, Now, Time
Durchführen von Datumsberechnungen	DateAdd, DateDiff, DatePart
Festlegen des Datums oder der Zeit	Date, Time
Verwenden eines Zeitgebers in einem Prozess	Timer
Zurückgeben einer Zeit	TimeSerial, TimeValue
Zurückgeben eines Datums	DateSerial, DateValue
Zerlegen eines Datums	Day, Month, Year, Weekday, WeekdayName, MonthName
Zerlegen einer Zeit	Hour, Minute, Second

Mathematik-Schlüsselwörter	
Ableiten trigonometrischer Funktionen	Atn, Cos, Sin, Tan
Abrufen des Absolutwertes	Abs
Abrufen des Vorzeichens eines Ausdrucks	Sgn
Allgemeine Berechnungen	Exp, Log, Sqr
Durchführen von numerischen Umwandlungen	Fix, Int
Erzeugen von Zufallszahlen	Randomize, Rnd
Runden	Round

Finanzmathematische Schlüsselwörter	
Berechnen der Abschreibung	DDB, SLN, SYD
Berechnen der Anzahl der Zeiträume	Nper
Berechnen des aktuellen Werts	NPV, PV
Berechnen des internen Zinsflusses	IRR, MIRR
Berechnen des Zinssatzes	Rate
Berechnen des zukünftigen Werts	FV
Berechnen von Zahlungen	Ipmt, Pmt, Ppmt

Eingabe-, und Ausgabe-Schlüsselwörter	
Abrufen von Informationen über eine Datei	EOF, FileAttr, FileDateTime, FileLen, FreeFile, GetAttr, Loc, LOF, Seek

Festlegen der Schreib-/Leseposition in einer Datei	Seek
Festlegen oder Abrufen von Dateiattributen	FileAttr, GetAttr, SetAttr
Kopieren einer Datei	FileCopy
Lesen aus einer Datei	Get, Input, Input #, Line Input #
Schließen von Dateien	Close, Reset
Schreiben in eine Datei	Print #, Put, Write #
Steuern des Darstellung der Ausgabe	Format, Print, Print #, Spc, Tab, Width #
Verwalten von Dateien	Dir, Kill, Lock, Unlock, Name
Zugreifen auf eine Datei oder Erstellen einer Datei	Open
Zurückgeben der Dateilänge	FileLen

Verzeichnis- und Datei-Schlüsselwörter	
Ändern des Laufwerks	ChDrive
Ändern des Verzeichnisses oder Ordners	ChDir
Entfernen eines Verzeichnisses oder Ordners	RmDir
Erstellen eines Verzeichnisses oder Ordners	MkDir
Festlegen von Attributinformationen für eine Datei	SetAttr
Kopieren einer Datei	FileCopy
Umbenennen einer Datei, eines Verzeichnisses oder eines Ordners	Name
Zurückgeben der Dateilänge	FileLen
Zurückgeben der Datums-/Zeitangabe einer Datei	FileDateTime
Zurückgeben des aktuellen Pfads	CurDir
Zurückgeben des Dateinamens oder der Datenträgerbezeichnung. Eignet sich um Dateien in einem Verzeichnis aufzulisten.	Dir
Zurückgeben von Datei-, Verzeichnis- oder Bezeichnungsattributen	GetAttr

Schlüsselwörter verschiedener Bereiche	
Abspielen eines Klangs vom Computer	Beep
Ausführen anderer Programme	AppActivate, Shell
Automatisierung	CreateObject, GetObject
Farbe	QBColor, RGB
Senden von Tastenanschlägen an eine Anwendung	SendKeys
Stellt eine Befehlszeilenzeichenfolge zur Verfügung	Command
System-Variablen	Environ
Verarbeiten ausstehender Ereignisse	DoEvents
Überprüfen von Datentypen	IsArray, IsDate, IsEmpty, IsError, IsMissing, IsNull, IsNumeric, IsObject
Benutzer Interaktion	MsgBox, InputBox

3.2 DAS EXCEL-OBJEKTMODELL

Das Excel-Objektmodell ist nichts anderes als eine Abbildung von Excel und seiner Elemente in Klassen der Programmiersprache VBA. Man kann sich das Objektmodell als hierarchische

Anordnung der Objekte vorstellen, wobei ein Objekt Teil eines anderen Objektes ist. Auf ein untergeordnetes Objekt kann man als Eigenschaft (oft in Form einer Auflistung) des übergeordneten Objektes zugreifen (oft auch umgekehrt, Eigenschaft `Parent`).

Das oberste Objekt in Excel ist die Anwendung Excel selber: das `Application`-Objekt. Die offenen Arbeitsmappen sind in der Eigenschaft `Workbooks` aufgelistet. Das `Workbook`-Objekt enthält dann eine Auflistung aller Arbeitsblätter in der `Sheets`-Eigenschaft. Elemente dieser Auflistung können entweder Tabellenblätter (`Worksheet`) oder Diagrammblätter (`Chart`) sein. In einem Tabellenblatt gibt es verschiedene Zellbereiche (`Columns`, `Rows`, `Cells`, `Range`), die selbst wiederum Auflistungen von Zellen (`Cells`) enthalten. Alle Zellen und Zellbereiche sind `Range`-Objekte, für die Eigenschaften wie Wert (`Value`), Formel (`Formula`) und Formatierung (Hintergrund (`Interior`), Schrift (`Font`) und Rahmen (`Border`)) definiert sind.

Beim Schreiben von Objekten und Eigenschaften helfen Autovervollständigung und Tooltips des VBA-Editors. Im Menü "Bearbeiten" finden Sie die Hilfsmittel: für die Tooltips sind das QuickInfo (Strg-I) und ParameterInfo (Strg-Umschalt-I).

Weiterführend: Für viele Objekte gibt es sog. Ereignisse, mit deren Hilfe man Programme schreiben kann, die etwa darauf reagieren, dass ein neues Arbeitsblatt eingefügt wird.

Übersicht über wichtige Objekte und Auflistungen:

- Application — Excel selbst
  - Workbook(s) — Arbeitsmappen
    - Worksheet(s) — Tabellenblätter
      - Cells — Zellen (Auflistung)
      - Rows — Zeilen (Auflistung)
      - Columns — Spalten (Auflistung)
      - Range — Zellen oder Zellbereiche
        - Cells, Rows, Columns
        - Range
      - ChartObjects — eingebettete Diagramme
        - Chart — Diagramm
      - Shape(s) — Zeichnungsebenen
    - Chart(s) — Diagrammblätter
    - Window(s) — Fenster

3.2.1 APPLICATION-OBJEKT

Das Application-Objekt ist Excel als laufende Anwendung. Es ist der Ausgangspunkt aller Referenzen und muss deshalb häufig nicht explizit angegeben werden.

Wichtig sind vor allem seine Auflistungen und speziellen Instanzen:

Workbooks: Auflistung aller offenen Arbeitsmappen.

Mit dem Namen oder dem Index der Mappe können Sie auf eine bestimmte Arbeitsmappe zugreifen: `Workbooks("test.xlsx")` oder `Workbooks(1)`. (Das ist die `Item`-Eigenschaft der Auflistung).

Mit For-Each können Sie durch alle Arbeitsmappe iterieren:

```
Dim eineMappe As Workbook
For Each eineMappe In Application.Workbooks
   Debug.Print eineMappe.Name
Next
```

Methoden und Eigenschaften:

- Eine Arbeitsmappe öffnen: `Workbooks.Open pfad`
  `Set eineMappe = Worksboos.Open(pfad)`
- Eine neue Arbeitsmappe erstellen: `Workbooks.Add`
  `Set neueMappe = Workbooks.Add`
- Alle Arbeitsmappe schliessen: `Workbooks.Close`
- Anzahl geöffneter Arbeitsmappe: `Workbooks.Count`

WorksheetFunction: Mit diesem Objekt können Sie die Arbeitsblattfunktionen in VBA benutzen. Die einzelnen Funktionen sind dessen Methoden:

```
WorksheetFunction.Sum(arg1, arg2)
```

Aktive Instanzen: Das Application-Objekt bietet Zugriff auf eine Reihe von Instanzen gerade aktiver Objekte:

- `ActiveWorkbook`: Die aktuelle Arbeitsmappe
- `ActiveWindow`: Das aktive Fenster in Excel
- `ActiveSheet`: Die aktive Arbeitsmappe (Tabelle oder Diagramm)
  - Ob Diagramm oder Tabelle können sie feststellen mit:
    `TypeName(ActiveSheet) = "Chart"` (bzw. "`Worksheet`")
- `ActiveChart`: Das gerade aktive eingebettete Diagramm
  - Wenn kein Chart aktiv ist, würde der Zugriff zu einem Fehler führen. Mit `ActiveChart Is Nothing` kann man das überprüfen. Etwa so:
    ```
    If Not (ActiveChart Is Nothing) Then
        MsgBox ActiveChart.Name
    End If
    ```
- `ActiveCell`: Die Zelle mit dem Cursor
- `Selection`: Gibt das gerade aktive Objekt in Excel zurück. Dies kann ein Zellbereich, ein Bestandteil eines Diagrammes oder ein anderes Objekt (wie ein Textfeld) sein. Mit `TypeName(Selection)` lässt sich dies eruieren.
- Mit `Columns`, `Rows` und `Cells` (u.a.) können Sie direkt auf die entsprechenden Auflistungen von Zellen des aktiven Tabellenblattes zugreifen.
- `Range(arg1[,arg2])`. Ein Zellbereich der aktiven Arbeitsmappe. Das Argument arg1 gibt den Zellbereich in A1 schreibweise an. Das zweites optionales Argument arg2 gibt die Zelle unten rechts im Bereich an.
  `Range("A1")`, `Range("A1:C4")` oder `Range("A1","C4)`

Das Application-Objekt besitzt sehr viele weitere Eigenschaften und Methoden. Nur zwei Beispiele, die beim Makroprogrammieren sinnvoll sein können:

- `DisplayAlerts`: damit können Sie bestimmte Warnungen und Meldungen unterdrücken, die Excel während der Ausführung des Markos anzeigen würde. Möchten Sie etwa eine Mappe schließen, würde Excel fragen, ob sie gespeichert werden soll:
```
Application.DisplayAlerts = False
Workbooks("Mappe1").Close
Application.DisplayAlerts = True
```
- `Calculation = xlCalculationManual`: stellt die automatische Berechnung der Arbeitsblattformeln aus. Mit `Calculation = xlCalculationAutomatic` wird sie wieder eingestellt.

Tipp: "`xlCalculationManual`" und "`xlCalculationAutomatic`" sind in VBA definierte Konstanten. Solche Konstanten helfen Ihnen Eigenschaften die richtigen Werte zuzuweisen. Sie folgen einem Muster: "xl" für Excel, "Calculation" für den Eigenschaftsname und "Manual" für den Wert. Mit Strg-Umschalt-J oder Menü: Bearbeiten: "Konstanten anzeigen" kann man sie sich anzeigen lassen (nach dem Gleichheitszeichen).

Wichtige Methoden:

- `Calculate`: Berechnet alle Arbeitsmappen (gibt es auch für Worksheet und Range)
- `OnTime(Zeit, Prozedurname, SpätesteZeit, Neu)`: Startet eine Methode zu einem bestimmten Zeitpunkt. Wenn `Neu` False ist, dann wird eine vorige OnTime-Methode gelöscht.

3.2.2 WORKBOOK-OBJEKT

Ein `Workbook` ist eine Arbeitsmappe. Verfügbar als Element der `Workbooks`-Auflistung (Item, Add oder Open) oder als aktive Arbeitsmappe (`ActiveWorkbook`).

Wichtige Eigenschaft

- `Sheets`: Auflistung aller Arbeitsblätter
- `Charts`: Auflistung aller Diagrammblätter
- `Worksheets`: Auflistung aller Tabellenblätter
  - Die Item-Methode gibt mit Index oder Namen ein bestimmtes Tabellenblatt zurück
  - ähnlich wie bei Workbooks gibt es hier: `Add`, `Delete` (alle!)
- `Windows`: Auflistung aller Fenster der Arbeitsmappe
- `Password`: Passwort der Arbeitsmappe
- `ReadOnly`: Ist die Arbeitsmappe schreibgeschützt?

Wichtige Methoden

- `Activate`: Aktiviert die Arbeitsmappe
- `Close`: Schliesst die Arbeitsmappe

- `Protect`: Schützt vor unberechtigten Änderungen durch Benutzer
- `Unprotect`: Hebt Schutz wieder auf
- `Printout`: Druckt die Arbeitsmappe
- `Save`: Speichert die Arbeitsmappe
- `SaveAs`: Speichert die Arbeitsmappe unter einem neuen Namen

3.2.3 WORKSHEET-OBJEKT

Ein Worksheet-Objekt entspricht einem Tabellenblatt. Verfügbar als Elemente der `WorkSheets`- und `Sheets`-Auflistung. Mit dem Tabellennamen oder dem Index kann eine bestimmte Tabelle der Auflistung abgerufen werden:

```
Dim wshTabelle as Worksheet
Set wshTabelle = Worksheets("Tabelle1")
Set wshTabelle = Worksheets(2)
```

Für das aktive Arbeitsblatt gibt es die spezielle Instanz `ActiveSheet`. `ActiveSheet` ist selbst als Objekt definiert (da es auch ein Chart-Objekt sein könnte). Einziger Nachteil: Keine Autovervollständigung der Methoden.

Wichtige Eigenschaften:

- `Rows` Auflistung der Zeilen des Tabellenblattes (Klasse Range)
- `Columns` Auflistung der Spalten des Tabellenblattes (Klasse Range)
  - Rows und Columns können Sie mit dem Index oder dem Namen (Spaltenbuchstaben oder Zeilenzahl in Anführungszeichen) ansprechen.
  - Spezielle Methoden:
    - `Insert`: Spalte/Zeile einfügen
      `ActiveSheet.Rows("7").Insert`
    - `Delete`: Spalte/Zeile löschen
    - `Hide`: Spalte/Zeile verbergen
- `Cells` Auflistung der Zellen des Worksheets (Klasse Range):
  Ohne Argument sind das alle Zellen der Tabelle.
  Mit zwei Argumenten ist es eine bestimmte Zelle:
  `Cells(zeile, spalte)`, wobei `zeile` und `spalte` die Indizes von Zeile/Spalte sind.
- `Range` gibt einen Zellbereich zurück. Das Argument gibt den Zellbereich in A1 schreibweise an. Ein zweites optionales Argument gibt die Zelle unten rechts im Bereich an.
  `Range("A1")`, `Range("A1:C4")` oder `Range("A1","C4)`
- `Shapes` Auflistung der Zeichenobjekte (Diagramme, Textboxen oder Bilder)
- `Name` Name des Tabellenblattes
- `ScrollArea` legt in "A1"-Schreibweise den für den Benutzer erreichbare Bereich des Blatts fest
- `Visible` ein- und ausblenden des Arbeitsblattes
- `ChartObjects` Auflistung der eingebetteten Diagramme

Wichtige Methoden

- `Activate` macht das Tabellenblatt zum aktiven Worksheet
- `Calculate` berechnet alle Formeln auf dem Tabellenblatt
- `Delete` löscht das Arbeitsblatt
- `Move` verschiebt das Tabellenblatt
- `Copy` kopiert das Tabellenblatt
- `PrintOut` druckt das Tabellenblatt
- `Select` wählt ein Tabellenblatt aus.
  - Sie können auch mehrere Blätter auswählen über die `Worksheets`-Auflistung:
    ```
    Worksheets(Array(1,2,3)).Select
    ```
    - `Array`(1,2,3) erstellt ein Datenfeld mit den Elemente 1,2 und 3.

3.2.4 RANGE-OBJEKT

Das Range-Objekt stellt eine Zelle oder einen Zellbereich dar. Es ist wohl eines der meist genutzten Objekte. Alle Auflistungen oder Eigenschaften, die irgendwie einen Teil der Tabelle darstellen (etwa Columns oder Cells), geben ein Range-Objekt zurück:

```
Dim EinBereich as Range
Set EinBereich = ActiveSheet.Columns("D")
Set EinBereich = ActiveSheet.Rows(3)
Set EinBereich = ActiveSheet.Rows("2:4")
Set EinBereich = Range("C4")
```

Die aktive Zelle ist mittels der speziellen Instanz `ActiveCell` oder `Selection` verfügbar (Achtung: `Selection` kann auch ein anderes Objekt sein):

```
Set EinBereich = Selection
```

Wichtige Eigenschaften:

- `Cells` Auflistung der Zellen im Bereich; Achtung: Range("E4:F7").Cells(1,1) ergibt Zelle E4. Gibt selbst ein Range zurück.
  `Cells(reihe, spalte)`
  `Cells(index)`: Zeile für Zeile
- `Rows` Auflistung der Zeilen im Bereich (Range)
- `Columns` Auflistung der Spalten im Bereich (Range)
- `Row`, `Column` Index der ersten Reihe, bzw. Spalte des Zellbereichs
- `EntireRow` Die Zeilen zu denen der Zellbereich gehört (Range)
- `EntireColumn` Die Spalten zu denen der Zellbereich gehört (Range)
- `Count` Anzahl Zellen im Zellbereich
- `Offset` gibt einen versetzten Bereich zurück (Range):
  `Offset(Row, Column)`
- `Locked` Zellen gesperrt oder nicht (True oder False)
- `Range` Einen Bereich (Range) relativ zum aktuellen Bereich (dabei ist "A1" die linke oberer Zelle)
- `Value` Wert der Zelle oder einen Array der Werte der Zellen im Zellbereich (lesen oder schreiben). (Standardeigenschaft von Range)
- `HasFormula` Ist in der Zelle eine Formel (Boolean)

- `Formula` Die Formel des Bereiches (Wert oder Array)
- `SpecialCells` Die Zellen eines bestimmten Types (etwas leere Zellen oder Zellen mit Formeln) (Range): `SpecialCells(xlCellType)`
- `Address` Bereichsbezug als String
- `Borders` Formatierung: Auflistung der Border-Objekte, die die Rahmenlinien definieren
- `Font` Formatierung: Font-Objekt, definiert die Textgestaltung
- `Interior` Formatierung: Hintergrund, Füllung der Zellen
- `Hyperlinks` Auflistung der Hyperlinks im Bereich (mit `Add` wird einer erstellt)
- `Worksheet` Tabellenblatt, in welchem sich der Bereich befindet

Wichtige Methoden

- `Select` Markiert den Bereich
- `Calculate` Berechnet alle Formeln im Bereich
- `Clear` Löscht alle Inhalte. Spezifische Inhalte löschen: `ClearComments, ClearContent, ClearFormat, ClearHyperlink`, usw.
- `Find` Einen Wert im Bereich suchen
- `Sort` Sortiert den Bereich
- `Delete` Löscht den Bereich
- `Insert` Fügt einen entsprechend großen Bereich an der Stelle ein (als Argument kann man die Verschieberichtung angeben)
- `AutoFilter` Schaltet den Autofilter für den Bereich ein (Filterkriterien als Argumente möglich).
- `Copy` Kopiert Bereich in die Zwischenablage
- `Paste` Fügt kopierten Bereich ein
- `Merge` Verbindet die Zellen des Zellbereichs
- `Unmerge` Hebt eine Zellverbindung auf
- `Show` Scrollt das Fenster so, dass der Zellbereich sichtbar ist.

Beispiel: einer Zelle einen Wert zuschreiben:

```
EinBereich.Value = "Irgend ein Text"
```

Beispiel: in einer Zelle einen Wert finden:

```
Function RangeWertFinden(strSuchtext As String _
                         , rngSuchOrt As Range) As String
    Dim rngFundort As Range
    Set rngFundort = rngSuchOrt.Find(strSuchtext)
    If rngFundort Is Nothing Then
        Err.Raise vbObjectError + 1 _
                , "WertSuchen" _
                , "Kein Wert gefunden"
    Else
        RangeWertFinden = rngFundort.Address
    End If
End Function
```

3.3 FORMULARE UND EREIGNISGESTEUERTE PROGRAMMIERUNG

In VBA kann man auch eigene Dialogfenster, sog. Formulare (Userforms) erstellen. Auf Formularen werden Steuerelemente, etwa Textfelder, Auswahllisten oder Schaltflächen platziert, die es dem Benutzer erlauben, die Ausführung des Programmes zu steuern.

Im VBA-Editor fügen Sie über das Menü "Einfügen":"Userform" ein neues Formular ein. Im Projektexplorer müsste dieses jetzt unter "Formulare" sichtbar sein, mit dem Kontextmenü können Sie das Objekt (das UserForm-Fenster) oder den Code (das Objektmodul) anzeigen lassen.

Die Steuerelemente (Controls) finden Sie in der Werkzugsammlung. Sie können dort ausgewählt und auf der Userform platziert werden. Userform und Controls sind in VBA Objekte. Das Spezielle an der Userform ist, dass Sie selbst weitere Eigenschaften und Methoden definieren, indem Sie in seinem Objektmodul Code schreiben.

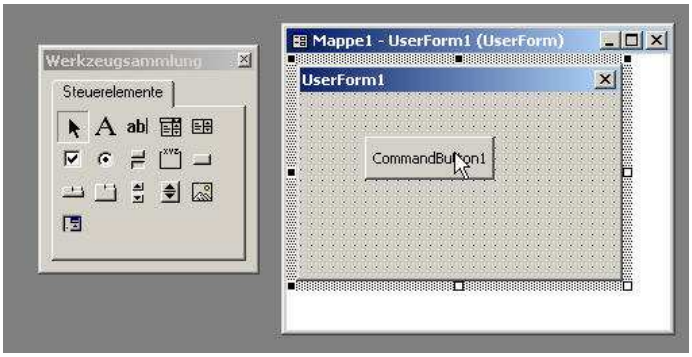


Abbildung 11: Userform-Fenster und Werkzeugsammlung

In Excel kann auch eine Arbeitsmappe als Formular verwenden werden. Auf dem Register "Entwicklungstools" unter "Einfügen" stehen hierfür die ActiveX-Steuerelemente zur Verfügung. Um Steuerelemente in Excel einzufügen und zu bearbeiten, muss man in den sog. "Entwicklungsmodus" wechseln. Auch für die Arbeitsmappe wird dann Code im Objektmodul geschrieben.

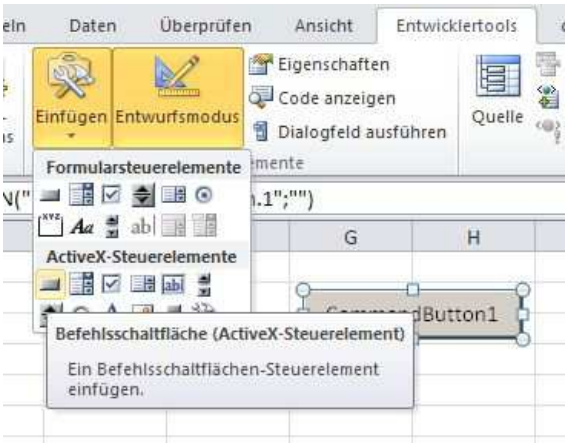


Abbildung 12: Arbeitsblatt und ActiveX-Steuerelemente

Die Eigenschaften der Userform oder eines Steuerelements können Sie bequem im Eigenschaftsfenster anpassen. Hier sollten Sie immer auch den Objektvariablen-Namen "(Name)" anpassen. Mit diesem Namen können Sie im Programmcode Eigenschaften des Steuerelementes ändern und dessen Methoden benutzen.

Wichtige Eigenschaften:

- `(Name)` Objektname des Controls, zur Verwendung im Code
- `Caption` Im Steuerelement dargestellter Text
- `Value` Vom Benutzer eingegebener Wert
- `Enabled` Control aktivieren oder desaktivieren
- `Visible` Control anzeigen oder verbergen

Methoden der Userform

- `Show` Formular anzeigen
- `Hide` Formular ausblenden (aber nicht entladen)

Im Objektmodul wird die Userform selbst mit dem Schlüsselwort **`Me`** referenziert. `Me.Caption="Mein Formular"` würde dessen Titel festlegen.

Anzeigen eines Formulars: Angezeigt wird ein Formular, wenn eine Prozedur die Show-Methode des Formulars aufruft:

```
Name_der_Userform.Show
```

Zum **Entladen eines Formulars** verwendet man die Anweisung:

```
Unload Name_der_Userform|Me
```

Wird die Unload-Anweisung nicht aufgerufen, entlädt VBA das Formular automatisch am Ende der Lebensdauer der Objektvariablen. In der Regel werden Sie mit dem Namen der Userform direkt arbeiten, man kann aber auch explizit eine Objektvariable definieren:

```
Dim meinFormular As UserForm
Set meinFormular = Name_der_Userform
```

Formular starten: Im VBA-Editor starten Sie ein Userform gleich wie eine Methode in einem normalen Modul (Menü:"Ausführen" oder mit F5). Um ein Formular in Excel als Makro verwenden zu können, muss in einem normalen Modul eine Methode ohne Argumente geschrieben werden, die das Formular mit der Show-Methode anzeigt:

```
Sub AnzeigenForm()
    MeinUserForm.Show
End Sub
```

Im Beispiel wird das Formular "MeinUserForm" angezeigt. Die Ausführung des Codes wird erst wieder fortgesetzt, wenn der Benutzer das Formular schließt. Während ein Formular angezeigt wird, ist auch Excel blockiert. Um dies zu vermeiden, kann man in der Showanweisung als Argument die Konstante `vbModeless` angeben:

```
Form.Show vbModeless
```

Dies bewirkt, dass das Formular angezeigt wird, die Ausführung des Codes aber direkt danach fortesetzt wird. Achtung: es kann dabei zu fehleranfälligen Überschneidungen von manuellen und automatischen Aktionen kommen!

Ereignisgesteuerte Programmierung: Benutzeraktionen in Formularen werden durch sog. Ereignisse gesteuert. Nachdem ein Formular angezeigt wurde, wartet VBA, bis der Benutzer etwas mit dem Formular tut. Wenn er etwa eine Schaltfläche drückt, ruft VBA im Objektmodul die Prozedur auf, die als Empfänger dieses Ereignisses (ein sog. Listener oder eine Callback-Funktion) definiert wurde.

Die allgemeine Form einer Ereignis-Prozedur (eines Listener) ist der Name des Steuerelements gefolgt vom Namen des Ereignisses getrennt durch einen Unterstrich. Je nach Ereignis werden auch Argumente übergeben:

```
Private Sub MeinControl_Ereignis([argumente])

End Sub
```

Im VBA-Editor muss man die Prozeduren nicht selber definieren. Ein Doppelklick auf das Steuerelement im Entwurfsmodus erstellt die Prozedur für das Standardereignis. Andere Ereignisse können Sie im Codefenster in den zwei oberen DropDowns auswählen: Im ersten wählen Sie das Steuerelement (inkl. der Userform selber), im zweiten das Ereignis, welches sie abfangen möchten. In Fettschrift erscheinen bereits vorhandene Ereignisprozeduren:

Abbildung 13: Ereignisprozeduren erstellen

Nicht nur die Steuerelemente sondern auch die Arbeitsmappe, die Arbeitsblätter und eine Userform selbst lösen Ereignisse aus. Sie werden ebenso verwendet wie Ereignisse der Controls, indem eine Ereignisprozedur im Objektmodul definiert wird. Für Userforms interessant sind vor allem: `UserForm_Initialize` und `UserForm_Terminate`. Damit können Sie Code schreiben, der ausgeführt wird bevor das Formular angezeigt bzw. beendet wird.

Tipp: Obschon der Benutzer eine Userform immer durch das Kreuz oben links schließen kann, empfiehlt es sich, eine Schaltfläche einzufügen, die das Formular beendet.

```
Private Sub buttonFensterSchliessen_Click()
    Unload Me
End Sub
```

Damit der Benutzer dies auch mit der Esc-Taste machen kann, können Sie die `Cancel`-Eigenschaft der Schaltfläche auf `True` setzen. Entsprechend gibt es auch eine `Default`-Eigenschaft, die bewirkt, dass die Eingabe-Taste das Click-Ereignis auslöst.

4 Ein Anwendungsbeispiel: Ein Formular um eine Liste in einer Tabelle zu füllen

Aufgabe:

Es soll ein Formular programmiert werden, mit dem Werte zu einer Liste auf einem Tabellenblatt hinzugefügt werden können. Auf dem Formular soll ein Optionskästchen sein, mit dem entschieden werden kann, ob alle Werte oder nur solche, die nicht in der Liste sind, zugefügt werden.

Das Formular soll mittels einer Schaltfläche auf der Tabelle gestartet werden. Allerdings nur dann, wenn eine Liste bzw. deren Überschrift in der Zelle A1 vorhanden ist. Ist dies nicht der Fall, soll der Benutzer aufgefordert werden, eine Liste neu zu erstellen oder den Vorgang abzubrechen.

Auf diesem Formular sollen ein Textfeld, ein OK-Button, ein Abbrechen-Button und ein Kontrollkästchen mit gesetztem Haken vorhanden sein.

Der Abbrechen-Button soll die Userform schließen.

Der OK-Button soll den Wert im Textfeld einfügen. Ist kein Wert im Textfeld, soll der Benutzer mit einer Meldung aufgefordert werden, einen Wert anzugeben.

Der Wert soll in die erste leere Zelle in der Spalte der Liste eingefügt werden. Ist die Option "nur eindeutige Werte" gewählt, muss überprüft werden, ob der Wert schon vorhanden ist.

Erweiterungen

- Wert geordnet einfügen
- Eine Fehlerbehandlung einbauen
- Im Formular eine Rückmeldung nach der Aktion anzeigen: ob und wo
- Mehrere Werte in einer Zeile einfügen
- Die Werte im Formular auflisten und die Option zum Löschen oder Bearbeiten anbieten
- Die Liste, um Werte einzufügen, in der Arbeitsmappe suchen.

Vorgehensweise:

Fügen Sie auf einem leeren Tabellenblatt eine ActiveX-Schaltfläche ein und nennen Sie diese "btnFormularStarten" (Eigenschaftsfenster im VBA-Editor oder über Kontextmenüpunkt "Eigenschaften"). Beschriften können Sie die Schaltfläche wie Sie wünschen.

Erstellen Sie die Ereignisprozedur für die Schaltfläche. Im Entwurfsmodus Doppelklick auf Schalfläche oder Kontextmenüpunkt "Code anzeigen". In der erstellten Prozedur überprüfen wir, ob die "Tabellenstruktur" passt ("Liste" in A1). Wenn dies der Fall ist, starten Sie ein Formular, sonst setzen Sie eine Warnmeldung ab.

Alternative: in einem neuen Modul die Prozedur als Makro (Sub ohne Argumente) schreiben und dann in der Tabelle eine normale Schaltfläche mit dem Makro verknüpfen.

Damit Sie den Formular-Aufruf schreiben können, müssen Sie erst ein Formular erstellen. Fügen Sie dem Arbeitsmappen-Projekt eine neue Userform ein und nennen Sie diese "frmWerteEinfügen".

Platzieren Sie auf dem Formular ein Textfeld "txtNeuerWert", einen Button "btnEinfügen", einen Button "btnAbbrechen" und ein Kontrollkästchen "ckbNurEindeutigeWerte". Sie können das Formular testweise mal starten, F5 oder übers Menü. Setzen Sie die Default-Eigenschaft des Einfüge-Buttons und die Cancel-Eigenschaft des Abbrechen-Buttons auf `true`.

Im Objektmodul des Formulars wird, um den Zielbereich übergeben zu können, eine öffentliche Objektvariable "gRngZielBereich" für einen Range deklariert:

```
Public gRngZielBereich as Range
```

Wir bearbeiten jetzt wieder die Start-Prozedur für das Formular. Definieren Sie eine Boolean-Variable. Diese wird gebraucht, um das Ergebnis der Überprüfung des Listenbereichs zu speichern. Bevor wir das Formular starten, müssen wir noch den Zielbereich übergeben. Um eine Instanz des Zellbereichs zu erhalten, verwenden wir die Eigenschaft Range des aktuellen Arbeitsblattes. Da diese auch global für die Applikation definiert ist, müssen wir "ActiveWorksheet" nicht explizit nennen. Das Formular starten wir mit vbModeless so, dass wir weiter in Excel arbeiten können:

```
Dim blnTabelleOk As Boolean
If blnTabelleOk Then
    Set frmWerteEinfügen.gRngZielBereich = Range("A1")
    frmWerteEinfügen.Show vbModeless
End If
```

Jetzt prüfen wir den Listenbereich (nach der Variablen-Deklaration). Da wir das Bereichsobjekt mehrfach verwenden, erstellen wir eine Variable, hier die Range-Eigenschaft einmal mit explizitem Worksheet-Objekt:

```
Dim blnTabelleOk As Boolean
Dim rngListe As Range
Set rngListe = ActiveSheet.Range("A1")

If rngListe.Value = "Liste" Then
    blnTabelleOk = True
End If
```

Falls der Bereich nicht der Listen-Titel ist, wird der Benutzer mit einer MsgBox gefragt, ob der Titel gesetzt werden soll. Die MsgBox kann mehrere Button und Symbole anzeigen und

gibt einen dem Button entsprechenden Wert zurück (das Argument "Buttons" erwartet eine Zahl, die mithilfe von verschiedenen Konstanten konstruiert werden kann; auch der Rückgabewert wird gegen eine VB-Konstante abgeglichen; nur zur Verdeutlichung mit benannten Argumenten). Die If-Anweisung wird um ein Else ergänzt:

```
If rngListe.Value = "Liste" Then
    blnTabelleOk = True
Else
    If MsgBox(Prompt:="Keine Liste, erstellen?" _
              , Buttons:=vbInformation + vbYesNo _
              , Title:="Achtung" _
             ) = vbYes Then
        rngListe.Value = "Liste"
        blnTabelleOk = True
    Else
        blnTabelleOk = False
    End If
End If
```

Man könnte hier die Zelle mit dem Titel noch formatieren. Der einfachste Trick hierfür ist es, ein Makro aufzuzeichnen und dann die entsprechenden Befehle für das "rngListe"-Objekt zu verwenden (im aufgezeichneten Code die Objektvariable "Selection" für die aktuelle Zelle durch "rngListe" ersetzen).

Der Code für den Formularstart sollte noch um eine Fehlerbehandlung ergänzt werden. Bei einem Fehler springen wir zur Sprungmarke "Fehler" und geben eine Meldung über den Fehler aus. "vbCr" fügt einen Zeilenumbruch vor der Fehlermeldung ein.

```
On Error GoTo fehler
Exit Sub
fehler:
    MsgBox "Formular kann nicht angezeigt werden" _
                    & vbCr & Err.Description _
                    , vbCritical, "Fehler"
```

Der ganze Code sieht dann so aus:

```
Private Sub btnFormularStarten_Click()
    Dim blnTabelleOk As Boolean
    Dim rngListe As Range
    On Error GoTo fehler
    Set rngListe = ActiveSheet.Range("A1")
    If rngListe.Value = "Liste" Then
        blnTabelleOk = True
    Else
        If MsgBox("Tabellenblatt nicht geeignet. " _
                                & "Dennoch fortfahren?" _
                                , vbInformation + vbYesNo _
                                , "Achtung" _
                                ) = vbYes Then
            rngListe.Value = "Liste"
            blnTabelleOk = True
        Else
            blnTabelleOk = False
        End If
    End If
    If blnTabelleOk Then
        Set frmWerteEinfügen.gRngZielBereich = rngListe
        frmWerteEinfügen.Show vbModeless
    End If

    Exit Sub
fehler:
        MsgBox "Formular kann nicht angezeigt werden" _
                        & vbCr & Err.Description _
                        , vbCritical, "Fehler"
End Sub
```

Fügen Sie nun die Ereignisprozeduren für die beiden Schaltflächen ein. In beiden Fällen verwenden wir das Klick-Ereignis "_Click":

Der Abbrechen-Button soll unser kleines Programm beenden, indem er das Formular entlädt. Da die `Unload` Anweisung im Objektmodul des Formulars selbst steht, können wir das Schlüsselwort `Me` als Objektvariable des Formulars auf sich selbst verwenden:

```
Private Sub btnAbbrechen_Click()
    Unload Me
End Sub
```

In der Klick-Prozedur für den Einfügen-Button prüfen wir erst, ob ein Wert eingegeben wurde, und suchen dann die erste leere Zelle, um diesen einzufügen.

Definieren Sie eine String-Variable und übergeben Sie dieser den Text der Textbox. Der eingegebene Text ist in der `Value`-Eigenschaft gespeichert:

```
    Dim strEinfügeWert As String
    strEinfügeWert = txtNeuerWert.Value
```

Wurde kein Wert eingegeben, soll der Benutzer mit einer `InputBox` aufgefordert werden, dies nachzuholen. Der neue Wert wird in die Textbox geschrieben. Ein leerer Rückgabestring der InputBox bedeutet, dass der Benutzer den Vorgang abgebrochen hat, die Prozedur wird mit `Exit Sub` verlassen:

```
If strEinfügeWert = "" Then
    strEinfügeWert = InputBox("Einzugügender Wert")
    If strEinfügeWert <> "" Then
        txtNeuerWert.Value = strEinfügeWert
    Else
        Exit Sub
    End If
End If
```

Um den Wert in die erste leere Zelle einzufügen, müssen wir die Liste durchsuchen. Dies tun wir mit einer Schleife über die ganze Spalte und der Funktion `IsEmpty`. Der übergebene Bereich ist aber nur die Zelle mit dem Listentitel. Die Eigenschaft `EntireColumn` gibt die ganze Spalte des Range-Objekts zurück. Selbst ein Range-Objekt können wir mit der Eigenschaft `Cells` auf alle Zellen der Spalte zugreifen und diese Auflistung mit einer `For each`-Schleife durchlaufen. Hierzu brauchen wir nochmals eine Range-Objektvariable "rngZelle". Wird eine leere Zelle gefunden, wird der Wert an die Value-Eigenschaft übergeben und die Schleife verlassen.

```
For Each rngZelle In gRngZielBereich.EntireColumn.Cells
    If IsEmpty(rngZelle) Then
        rngZelle.Value = strEinfügeWert
        Exit For
    End If
Next
```

Es muss dann noch geprüft werden, ob nur eindeutige Werte eingefügt werden sollen und ob der Wert schon vorhanden ist. Wir verknüpfen zwei Bedingungen mit dem logischen Operator "And":

```
CBool(ckbNurEindeutigeWerte.Value) And rngZelle.Value =
strEinfügeWert
```

Ergänzen Sie die obige If-Anweisung, indem Sie mit `ElseIF` eine doppelte Prüfung konstruieren:

```
    If CBool(ckbNurEindeutigeWerte.Value) _
            And rngZelle.Value = strEinfügeWert Then
        Exit For
    ElseIf IsEmpty(rngZelle) Then
        rngZelle.Value = strEinfügeWert
        Exit For
    End If
```

Damit man bequem wiederholt neue Werte eingeben kann, setzen wir den sog. Fokus im Formular (gewissermaßen dessen Einfügemarke) auf die Textbox und markieren dort den ganzen Text. Die Konstruktion With-End With strukturiert die Anweisungen für das Textbox-Objekt:

```
    With txtNeuerWert
        .SetFocus
        .SelStart = 0
        .SelLength = Len(txtNeuerWert.Value)
    End With
```

Ergänzt um eine kleine Fehlerroutine sieht die fertige Prozedur nun so aus:

```
Private Sub btnEinfügen_Click()
    Dim strEinfügeWert As String
    Dim rngZelle As Range
    On Error GoTo fehler
    strEinfügeWert = txtNeuerWert.Value
    If strEinfügeWert = "" Then
        strEinfügeWert = InputBox("Einzugügender Wert")
        If strEinfügeWert <> "" Then
            txtNeuerWert.Value = strEinfügeWert
        Else
            Exit Sub
        End If
    End If

    For Each rngZelle In gRngZielBereich.EntireColumn.Cells
        If CBool(ckbNurEindeutigeWerte.Value) _
                And rngZelle.Value = strEinfügeWert Then
            Exit For
        ElseIf IsEmpty(rngZelle) Then
            rngZelle.Value = strEinfügeWert
            Exit For
        End If
    Next

    With txtNeuerWert
        .SetFocus
        .SelStart = 0
        .SelLength = Len(txtNeuerWert.Value)
    End With
Exit Sub
fehler:
    MsgBox "Fehler: " & Err.Description, vbCritical
    Unload Me
End Sub
```

Um das Programm aus Excel zu verwenden, müssen Sie nur noch den Entwicklungsmodus verlassen. Starten Sie das Makro durch einen Klick auf die Schaltfläche. Jetzt beginnt der schwierigste Teil der Programmierung: Testen und Debuggen. Es muss getestet werden, ob die gewünschten Resultate erzeugt werden. Mögliche Bedingungen müssen durchprobiert werden, um Fehler zu finden.