
6
4,40
Deutschland
KnowWare
Management!
Programmieren lernen mit Excel-VBA 5-6
KnowWare Management!
VBA mit Excel
... für Version 2000-2003
ENTER
V
B
A
Helma Spona
www.KnowWare.de
Deutschland: 4,40 EUR Österreich: 5,00 EUR
Schweiz: 8,60 SFR Luxemburg: 5,20 EUR
4 196468 404408 05006

Bildqualität

Wir versuchen die Dateigröße zu reduzieren, um die Downloadzeit zu verkürzen. Daher ist die Bildqualität in dieser Download-Datei nicht in allen Fällen optimal. Im Druck tritt dieses Problem nicht auf.

Acrobat Reader: Wie komme ich klar?

F5/F6 öffnet/schließt die Ansicht **Lesezeichen**
Strg+F sucht

Im Menü Ansicht stellst du ein, wie die Datei angezeigt wird
STRG+0 = Ganze Seite **STRG+1** = Originalgröße **STRG+2** = Fensterbreite

Im selben Menü kannst du folgendes einstellen: **Einzelne Seite**, **Fortlaufend** oder **Fortlaufend - Doppelseiten** ... Probiere es aus, um die Unterschiede zu sehen.

Navigation
Pfeil Links/Rechts: eine Seite vor/zurück
Alt+ Pfeil Links/Rechts: Wie im Browser: Vorwärts/Zurück
Strg+**+** vergrößert und **Strg**+**–** verkleinert

Bestellung und Vertrieb für den Buchhandel

KnowWare-Vertrieb, Postfach 3920, D-49029 Osnabrück
Tel.: +49 (0)541 33145-20 Fax: +49 (0)541 33145-33
bestellung@knowware.de
www.knowware.de

Autoren gesucht

Der KnowWare-Verlag sucht ständig neue Autoren. Hast du ein Thema, dass dir unter den Fingern brennt? - ein Thema, das du anderen Leuten leicht verständlich erklären kannst?

Schicke uns einfach ein paar Beispielseiten und ein vorläufiges Inhaltsverzeichnis an folgende Adresse:

lektorat@knowware.de

Wir werden uns deinen Vorschlag ansehen und dir so schnell wie möglich eine Antwort senden.

www.knowware.de

Inhaltsverzeichnis

Vorwort

Du hast ein Problem mit Excel, das sich nur mit Makros lösen lässt, oder du möchtest einfach programmieren lernen, ohne eine teure Programmiersprache kaufen zu müssen?

Dann bist du hier wirklich richtig. Dieses Heft soll dir die Grundlagen der VBA-Programmierung vermitteln – und zwar anhand von VBA 6.0, der in Excel 2000 und höher integrierten Programmiersprache. Viele Programmierer bezeichnen VBA zwar als „Programmiersprache für Arme“ – das ist aber keinesfalls richtig. VBA kann vieles von dem, was eine gute Programmiersprache kann, objektorientierte Programmierung eingeschlossen. Lass dir also nichts einreden.

VBA, die Abkürzung für Visual Basic für Anwendungen, ist die richtige Programmiersprache für alle, die eine Microsoft-Office-Anwendung wie Word, Excel oder Access haben und diese steuern und manipulieren möchten. Außerdem ist VBA recht einfach zu erlernen, sodass du damit auch sehr anschaulich die ersten Programmierschritte machen kannst.

Anforderungen

Willst du erfolgreich mit diesem Heft arbeiten, solltest du die Grundlagen von Windows kennen und wissen, wie du Dateien erstellst, kopierst und verschiebst. Du solltest dich auch so weit mit Excel 2000 (oder höher) auskennen, dass dir Begriffe wie Zelle, Zelladresse und Tabellenblatt bekannt sind.

Außerdem brauchst du natürlich noch Software. Da VBA in Excel integriert ist, benötigst du nur Excel 2000 (oder höher) für Windows.

Auch in Excel 2001 und höher für Macintosh ist VBA enthalten – allerdings erst in der Version 5.0. Das gilt auch für die ganz neue Version Excel 2004 für Mac OS X. Zwar werden einige Beispiele dann nicht funktionieren – die Grundlagen der VBA-Programmierung kannst du aber auch damit erlernen.

Grenzen

Auch wenn du Excel bzw. Office schon installiert hast, solltest du die Installations-CD bereithalten; die VBA-Hilfe wird nämlich standardmäßig nicht mit installiert – willst du sie benutzen, musst du sie also nachträglich installieren. Ohne diese Hilfe wirst du nämlich Schwierigkeiten haben, auch nur kleine Anwendungen zu erstellen – aus Platzgründen kann dieses Heft leider nur die Grundlagen liefern und dir zeigen, wie die VBA-Hilfe dir – nun ja: weiterhilft.

Am Ende des Heftes solltest du dich aber so gut auskennen, dass du dich fortbilden kannst, ohne in teure Literatur investieren zu müssen. Die Hilfe enthält nämlich alles, was man zu VBA wissen muss. Es geht also vor allem darum, die in der Hilfe verwendeten Begriffe zu verstehen – und genau das, oder die Grundlagen dazu, lernst du in den nachfolgenden Kapiteln.

Hilfestellung notwendig?

Solltest du mit einem Beispiel oder einer Erklärung aus diesem Heft nicht zurechtkommen, kannst du dich gerne an mich wenden.

Besuche dazu einfach meine Webseite:

http://www.helma-spona.de

Hier findest du ein Kontaktformular, über das du deine Frage loswerden kannst.

Ich bitte aber um Verständnis dafür, dass ich wirklich nur Fragen zu meinen Veröffentlichungen beantworte und keinen darüber hinausgehenden kostenlosen Support leisten kann.

Und nun viel Spaß und viel Erfolg beim Programmieren mit VBA!

Helma Spona

Die Entwicklungsumgebung

Jede Programmiersprache benötigt eine Entwicklungsumgebung. Dieser Begriff bezeichnet ein Programm, in dem du deinen Code erfassen, testen, und sofern notwendig, kompilieren kannst. Kompilieren bedeutet, dass der Quellcode, der aus einfachen Textanweisungen besteht, in eine Form überführt wird, die von einem Computer ausgeführt werden kann.

Das Programm, das die Kompilierung durchführt, heißt Compiler.

VBA-Code wird automatisch kompiliert, wenn du ihn ausführst. Ein separater Schritt ist dazu nicht erforderlich.

Allerdings führt die Kompilierung von VBA-Code nicht zu einer eigenständigen Anwendung, wie das eine EXE-Datei wäre, die du einfach per Doppelklick ausführen kannst. VBA-Code benötigt eine so genannte VBA-Hostanwendung. Dies kann prinzipiell jede Anwendung sein, die VBA in der entsprechenden Version unterstützt, also sowohl Word als auch Excel, PowerPoint oder Access.

Der VBA-Code wird grundsätzlich in der Datei der entsprechenden Hostanwendung gespeichert. Willst du den Code also in Excel ausführen, musst du ihn in eine Excel-Arbeitsmappe schreiben und kannst diese dann in Excel öffnen. Bearbeiten kannst du den Code in jeder VBA-Hostanwendung im gleichen Editor – dem VBA-Editor.

Verschiedene VBA-Versionen

Es gibt nicht nur eine Version von VBA. In Excel 5.0 und Excel 7.0, den ersten VBA-Hostanwendungen, war genau wie in Office 97 die Version 5.0 integriert. Alle folgenden Office-Anwendungen für Windows, also Office 2000, XP und 2003, unterstützen hingegen VBA 6.0 bzw. VBA 6.3 (Office 2003).

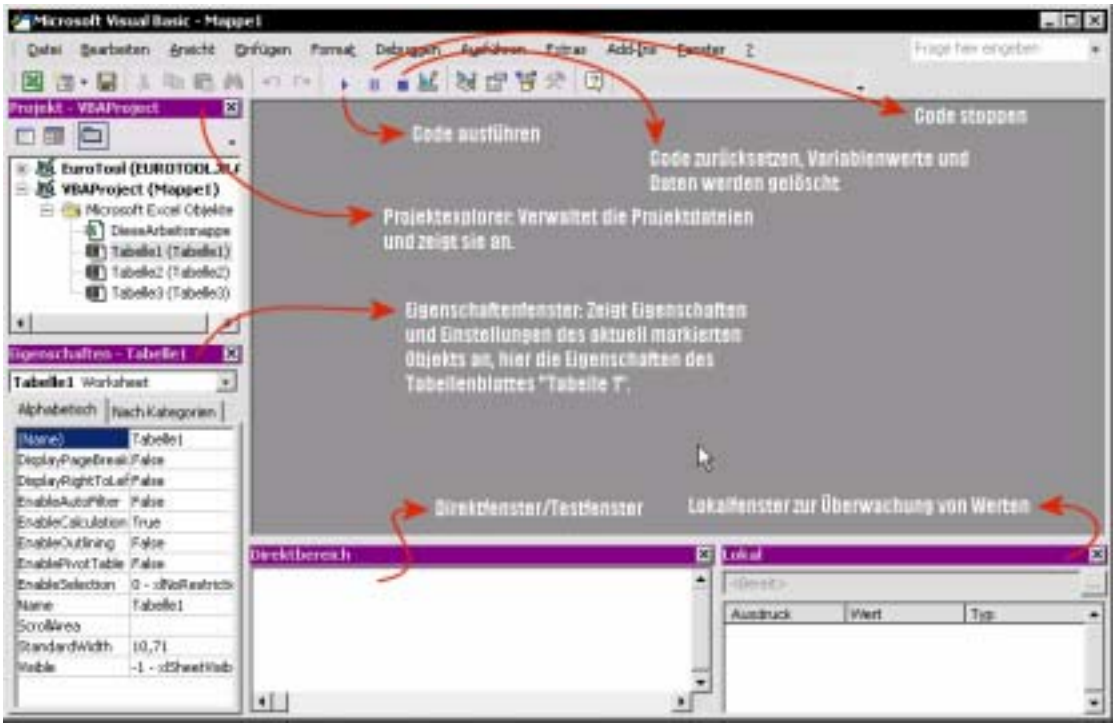
Auf dem Mac sieht das ein bisschen anders aus. Auch in der neuesten Office-Version, Office 2004 für Mac OS X, ist lediglich VBA 5.0 enthalten. Aber spielt das überhaupt eine Rolle?

Nun ja – auch mit VBA 5.0 kannst du VBA erlernen und nützliche kleine Anwendungen erstellen. Es gibt in VBA 6.0 jedoch einige Befehle, die in der älteren Version noch nicht verfügbar sind. Werden nachfolgend solche Befehle verwendet, wird darauf hingewiesen.

Den VBA-Editor starten

Um den VBA-Editor zu starten, musst du VBA nicht extra installieren – es wird automatisch mit der VBA-Hostanwendung installiert. Du kannst also gleich loslegen:

1. Starte Excel 2000 (oder höher).
2. Excel erzeugt nun automatisch eine leere Arbeitsmappe, die du gut für die ersten Schritte verwenden kannst. Falls du den Code in eine vorhandene Arbeitsmappe integrieren möchtest, öffnest du diese.
3. Drücke Alt + F11, um den VBA-Editor zu starten.



Die Entwicklungsumgebung im Überblick

Die Entwicklungsumgebung besteht aus mehreren Fenstern. Die wichtigsten sind links oben der Projektexplorer, links unten das Eigenschaftenfenster sowie das Direktfenster.

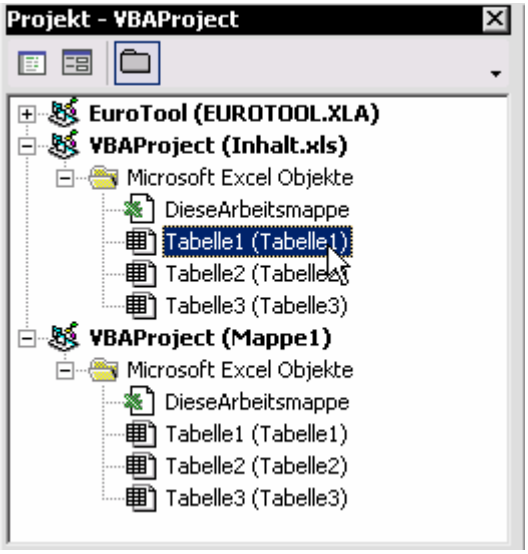
Alle Fenster der Entwicklungsumgebung kannst du über ANSICHT ein- oder ausblenden.

Im Eigenschaftenfenster kannst du die Eigenschaften von Tabellen, Formularen und anderen Elementen der Excel-Arbeitsmappe setzen oder einsehen. Näheres dazu folgt in den Abschnitten „Module erstellen und bearbeiten“ auf Seite 8 und „Benutzeroberflächen gestalten“ auf Seite 47. Im Direktfenster, auch Direktbereich oder Testfenster genannt, kannst du Befehle direkt eingeben und ausführen – du kannst aber auch Ausgaben zur Kontrolle der Anwendung machen lassen.

Der Projektexplorer

Mit dem Projektexplorer verwaltest du das VBA-Projekt. Hier kannst du auf Module zugreifen und diese manipulieren oder neue Module einfügen.

Module enthalten den VBA-Quellcode der Anwendung und werden als Eintrag im Projektexplorer angezeigt.



Der Projektexplorer mit zwei geöffneten Arbeitsmappen

Die Einträge im Projektexplorer werden als Baumstruktur dargestellt. Ganz oben steht immer der Name des VBA-Projektes, in Klammern

dahinter der Name der Arbeitsmappe – z.B. „Mappe1“ und „Inhalt.xls“. Auch geladene Add-Ins werden so angezeigt – etwa wie in der Abbildung das EuroTool-Add-In. Diesen Einträgen sind die Inhalte des VBA-Projekts untergeordnet. Sie sind nach Typen gruppiert. Die erste Gruppe fasst alle Excel-Objekte (in Word wären es Word-Objekte) zusammen. Jedes Excel-Tabellenblatt verfügt über ein Modul und wird daher hier aufgeführt. Enthält die Arbeitsmappe Diagramme, werden diese ebenfalls in der Gruppe „Microsoft Excel Objekte“ angezeigt.

Einfache Module, die du erzeugen kannst, um Code zu erstellen, der unabhängig von einem Tabellenblatt ist, werden in einer Rubrik „Module“ aufgeführt, die mit dem ersten Modul erzeugt wird. Darüber hinaus kann es Klassenmodule und UserForms (Formulare) geben. Beide werden später noch genauer erläutert.

Möchtest du ein Modul öffnen, das im Projektexplorer angezeigt wird, klickst du dazu einfach doppelt auf den Eintrag.

Es gibt VBA-Projekte, etwa die von Add-Ins, die geschützt sind. Falls du zur Kennworteingabe aufgefordert wirst, kannst du das Modul nur mit dem gültigen Kennwort öffnen; ansonsten wird der Inhalt des Moduls rechts neben dem Projektexplorer im Modulfenster angezeigt.

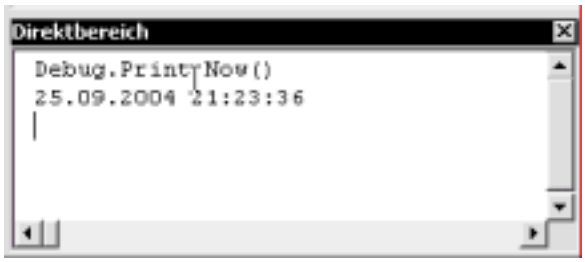
Anweisungen im Direktfenster ausführen

Das Direktfenster ist vor allem anfänglich ein sehr nützliches Hilfsmittel. Du kannst dort Anweisungen testen oder Werte ausgeben, die du zur Kontrolle des VBA-Codes benötigst.

Das wollen wir gleich testen:

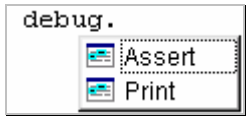
1. Setze den Cursor in das Direktfenster, indem du einfach hinein klickst.
2. Gib den Code `Debug.Print Now()` ein.
3. Drücke Enter.

Die Entwicklungsumgebung führt nun diesen Befehl aus. Zunächst wird die Anweisung `Now()` ausgewertet. Dabei handelt es sich um eine Funktion, die das aktuelle Datum und die aktuelle Uhrzeit zurückgibt. Dieser zurückgegebene Wert wird mit `Debug.Print` im Testfenster ausgegeben, was zur Folge hat, dass in der nächsten Zeile das Datum und die Uhrzeit erscheinen.



Der Befehl und seine Ausgabe im Direktbereich

Vielleicht hast du bei der Eingabe des Befehls gemerkt, dass nach dem Punkt ein kleiner Tooltipp erscheint. Dies ist die integrierte Programmierhilfe IntelliSense. Sie hilft dir bei der Eingabe des Codes, indem sie die verfügbaren Befehle zur Auswahl anbietet. Du kannst mit der ⇆-Taste und den Pfeiltasten einen Eintrag wählen und ihn mit Enter in den Code übernehmen.



Die Elementliste der Programmierhilfe

Neben der Elementliste gibt es noch weitere Programmierhilfen, wie z.B. die Parameterliste. Sie wird im Abschnitt „Module erstellen und bearbeiten“ auf Seite 8 näher erläutert.

Die VBA-Hilfe aufrufen

Neben der Programmierhilfe liefert natürlich auch die VBA-Hilfe zusätzliche Informationen. Du kannst sie nur aus der Entwicklungsumgebung (= IDE) aufrufen. Im Excel-Anwendungsfenster erscheint beim Aufruf der Hilfe mit F1 die Hilfe zu Excel, nicht die zu VBA. Wenn du die VBA-Hilfe aufrufen möchtest, gibt es dazu zwei Möglichkeiten.

- Drückst du im VBA-Editor einfach F1, wird die allgemeine VBA-Hilfe angezeigt.
- Alternativ kannst du auch den Cursor in einen VBA-Befehl stellen und dann F1 drücken. In diesem Fall wird die Hilfe zu dem aktuellen VBA-Befehl angezeigt.

Explizites installieren der VBA-Hilfe

Sollte die Hilfe in beiden Fällen nicht angezeigt werden oder erhältst du sogar die Meldung, dass das Feature nicht installiert ist, folge den Anweisungen des Assistenten, um es zu installieren. Bei Excel 2000 musst du unter Umständen das Setup-Programm manuell von der CD starten und die Installation anpassen. Je nach deiner Office-Version findest du die VBA-Hilfe in der Gruppe GEMEINSAM GENUTZTE OFFICE-KOMPONENTEN oder in der Gruppe HILFEDATEIEN.

Willst du z. B. die Hilfe zum `Debug`-Befehl aufrufen, gehst du wie folgt vor:

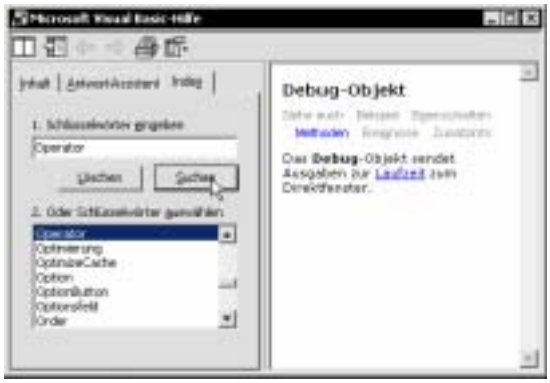
1. Gib im Direktbereich den Befehl `Debug` ein und setze den Cursor in das Wort. Steht der Befehl bereits im Testfester, kannst du auch einfach den Cursor in das vorhandene Wort setzen, ohne es neu einzugeben.
2. Drücke F1.

Die Hilfeseite zum Debug-Befehl

Über das Symbol EINBLENDEN der Symbolleiste kannst du das Inhaltsverzeichnis einblenden und so auch im Index suchen oder gezielt Themen der VBA-Hilfe anzeigen lassen.

Suchst du gezielt Informationen zu einem Begriff, der kein VBA-Befehl ist, funktioniert das wie folgt:

1. Öffne die VBA-Hilfe, indem du in der Entwicklungsumgebung F1 drückst.
2. Blende über das Symbol EINBLENDEN das Inhaltsverzeichnis ein.
3. Aktiviere die Registerkarte INDEX.
4. Gib in das Feld 1. SCHLÜSSELWÖRTER EINGEBEN den Suchbegriff ein.
5. Sollte es schon einen Indexeintrag geben, der dem Suchbegriff entspricht, wird er in der Liste darunter angezeigt und ausgewählt. Ein Doppelklick auf den Listeneintrag reicht aus, um die Hilfeseite zu öffnen. Sollte das Suchwort nicht als Indexbegriff definiert sein, klicke nun auf SUCHEN, um alle Hilfeseiten aufgelistet zu bekommen, die den Begriff enthalten. Auch dann kannst du per Doppelklick auf den Listeneintrag die entsprechende Hilfeseite öffnen.



Suchen nach einem Hilfe-Thema

Module erstellen und bearbeiten

VBA-Quellcode wird in Modulen gespeichert. Die maximale Anzahl Codezeilen in einem Modul ist begrenzt; allerdings wirst du erst bei sehr großen Anwendungen an diese Grenze stoßen. Für den Anfang können wir sie also vernachlässigen.

Grundsätzlich gibt es in einem VBA-Projekt zwei Typen von Modulen: einfache Module und Klassenmodule. Nicht alle Befehle sind in beiden Modultypen zulässig; bei den meisten ist es jedoch egal, in welchem Modultyp du sie speicherst. Einfache Module, der Name sagt es, stellen die einfachste Art von Modulen dar. Sie sind fast universell verwendbar.

Nachfolgend werden immer einfache Module verwendet – es sei denn, es wird explizit darauf hingewiesen, dass ein Klassenmodul notwendig ist.

Klassenmodule sind eine besondere Form von Modulen. Für jedes Tabellenblatt der Arbeitsmappe erzeugt Excel automatisch ein solches Klassenmodul, das du dann im Projektexplorer in der Gruppe MICROSOFT EXCEL OBJEKTE angezeigt bekommst. Aus diesen Klassenmodulen erzeugt Excel zur Laufzeit ein Objekt, das in diesem Fall das Tabellenblatt darstellt.

Du kannst aber auch eigene Klassenmodule erstellen und aus diesen Objekte ableiten. Klassenmodule entsprechen den Klassen anderer objektorientierter Programmiersprachen.

Du kannst dir eine Klasse wie eine Art Schablone vorstellen, mit der du gleichartige Objekte erstellen kannst.

Jedes Objekt, dass du aus einer Klasse erzeugst, hat alle durch die Klasse bestimmten Eigenschaften, unterscheidet sich aber dennoch von jedem anderen Objekt, das aus derselben Klasse abgeleitet wurde.

Die Ableitung eines Objektes aus einer Klasse wird Instanzierung genannt. Objekte werden alternativ auch als Instanzen bezeichnet.

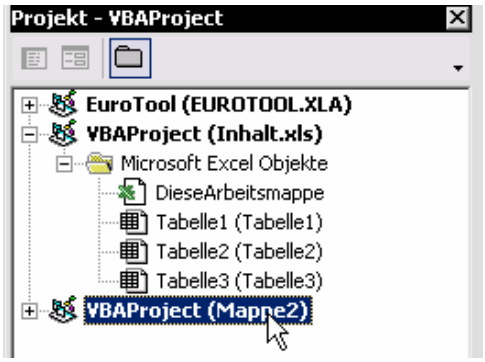
Nehmen wir an, du hast eine Schablone für Kreise. Dann bestimmt diese Schablone die Größe des Kreises. Jeder Kreis, den du mit der Schablone zeichnest, ist also gleich groß – und da es ein Kreis ist, sieht er damit auch immer gleich aus. Dennoch unterscheidet sich jeder gezeichnete Kreis von jedem anderen, mindestens durch die Position auf dem Blatt. In der Programmierung entspricht dies der Position im Hauptspeicher des Rechners, an der das Objekt gespeichert wird.

Klassenmodule benötigst du nur für ganz spezifische Vorhaben. Du wirst im Abschnitt „Benutzeroberflächen gestalten“ auf Seite 47 Näheres dazu erfahren.

Ein Modul erzeugen und benennen

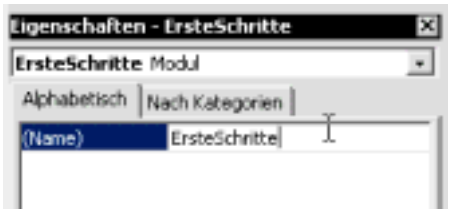
Willst du ein Modul erstellen, gehst du folgendermaßen vor:

1. Falls mehr als eine Arbeitsmappe geöffnet ist, markierst du im Projektexplorer die Arbeitsmappe, der du das Modul hinzufügen möchtest, indem du den entsprechenden Eintrag anklickst.



Aktivieren der Arbeitsmappe, der das Modul hinzugefügt werden soll

2. Wähle nun EINFÜGEN|MODUL.
3. Setze den Cursor in das Feld NAME des Eigenschaftenfensters.



Benennen des neuen Moduls

4. Überschreibe den vorhandenen Namen, z. B. „Modul1“ durch einen Namen deiner Wahl, z. B. „ErsteSchritte“.
5. Schließe die Eingabe mit Enter ab.

Damit hast du ein leeres Modul erzeugt und benannt. Wichtig ist, dass alle Module innerhalb eines Projektes einen eindeutigen Namen haben. Du kannst also nicht zwei oder mehr Modulen den gleichen Namen geben.

Ein Modulname darf keine Leer- und Sonderzeichen und auch keine Umlaute oder ein „ß“ enthalten. Lediglich ein Unterstrich ist als Sonderzeichen zulässig.

Code eingeben

Willst du in ein vorhandenes Modul Code eingeben, machst du das so:

1. Klicke im Projektexplorer doppelt auf das Modul, damit es als Modulfenster geöffnet wird.
2. Klicke an die Stelle im Modul, an der du den Code eingeben willst, z. B. an den Anfang des Moduls.

Nun kannst du den Code eingeben. Enthält das Modul noch keine Codezeile, solltest du ganz oben zunächst die Anweisung

```
Option Explicit
```

eingeben. Schließe die Eingabe mit Enter ab, damit der Cursor in die nächste Zeile springt. Diese Anweisung bewirkt, dass der VBA-Editor eine Fehlermeldung ausgibt, wenn eine Variable nicht deklariert ist.

Eine Variable ist ein benannter Wert, dessen Wert sich während des Programmablaufs ändern kann.

Wie du eine Variable deklarierst und was eigentlich eine Deklaration ist, erfährst du im Abschnitt „Code strukturieren“ auf Seite 12.

TIPP: Wünschst du, dass diese Anweisung in jedes neue Modul eingefügt wird, wählst du EXTRAS|OPTIONEN und aktivierst auf der Registerkarte EDITOR das Kontrollkästchen VARIABLENDEKLARATION ERFORDERLICH. Dann schließt du das Dialogfeld mit OK.

Alle ausführbaren Anweisungen werden in VBA innerhalb einer Prozedur oder Funktion definiert. Um die feinen Unterschiede brauchst du dich zurzeit noch nicht zu kümmern. Sowohl Prozeduren als auch Funktionen sind benannte Codeblöcke, die du ausführen kannst, indem du ihren Namen im Direktfenster eingibst oder den Cursor in die Prozedur (oder Funktion) setzt und dann **F5** drückst.

Willst du eine solche Prozedur erstellen, hast du zwei Möglichkeiten. Für den Anfang verwendest du am einfachsten das Menü.

1. Wähle EINFÜGEN|PROZEDUR.
2. Gib in das Feld NAME den Namen für die Prozedur ein, also z. B. „Meldung“.
3. Schließe das Dialogfeld mit OK.

Erstellen einer Prozedur mit Hilfe des Dialogfelds

Ein Prozedurname darf keine Leer- und Sonderzeichen und keine Umlaute oder ein „ß“ enthalten. Lediglich ein Unterstrich ist als Sonderzeichen zulässig. Zudem muss der Name innerhalb eines Moduls eindeutig sein.

Damit hast du nun eine Prozedur erstellt – und die IDE setzt den Cursor automatisch in die Prozedur. Du kannst nun den Inhalt der Prozedur direkt eingeben. Mit der `MsgBox`-Anweisung kannst du z. B. eine Meldung ausgeben und dabei gleich die automatische Parameterliste testen.

Ein Parameter ist ein Wert, der an eine Prozedur oder einen VBA-Befehl übergeben wird.

Gehe dazu folgendermaßen vor:

1. Gib den Text `MsgBox` gefolgt von einem Leerzeichen ein.
2. Die IDE blendet nun die Parameterliste ein. Sie zeigt die Parameter des Befehls in der notwendigen Reihenfolge an.

Die Parameterliste zum Befehl `MsgBox`

3. Der erste angezeigte Parameter heißt „Prompt“ und wird fett dargestellt. Das bedeutet, dass du diesen Parameter als nächstes eingeben musst. Gib dazu in Anführungszeichen den gewünschten Text für die Meldung ein.

4. Gib anschließend ein Komma ein. Das Ergebnis: Der nächste Parameter „Buttons“ wird fett dargestellt. Außerdem öffnet die IDE die automatische Konstantenliste mit den für diesen Parameter definierten Werten. Der Parameter bestimmt die Symbole und Schaltflächen, die in der Meldung angezeigt werden sollen.
5. Wähle in der Liste den Eintrag VBINFORMATION. Dazu kannst du „vbIn“ eingeben; der Eintrag wird dann markiert und du kannst ihn mit der ⇆-Taste übernehmen. Alternativ könntest du auch mit der Maus oder den Pfeiltasten durch die Liste scrollen und doppelt auf den gewünschten Eintrag klicken.
6. Drücke Enter, um die Eingabe abzuschließen und in die nächste Zeile zu springen.

Damit hast du eine korrekte Prozedur erstellt. Sie sollte nun folgendermaßen aussehen – wobei der Text "Meldung" die auszugebende Meldung darstellt und damit auch anders lauten darf:

```
Public Sub Meldung()
    MsgBox "Meldung", vbInformation
End Sub
```

Code ausführen

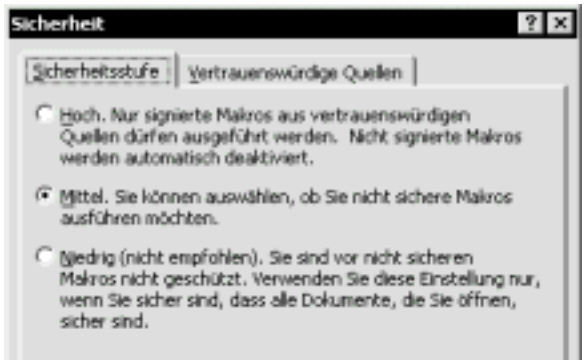
Nun kannst du den Code ausführen. Vorher solltest du den Code aber noch speichern, indem du das entsprechende Symbol in der Symbolleiste anklickst.

Du solltest den Code grundsätzlich speichern, bevor du ihn nach einer Änderung das erste Mal ausführst. Abstürze durch Codefehler können sonst zu einem Datenverlust führen.

Makrosicherheit auf Mittel setzen

Aus Sicherheitsgründen setzt Excel die Makrosicherheit nach der Installation auf „hoch”. Das bedeutet, dass Makros deaktiviert sind. Willst du den Code testen, musst du die Makrosicherheit prüfen und gegebenenfalls wie folgt auf mittel setzen:

1. Wechsle über die Taskleiste von Windows zum Excel-Fenster.
2. Wähle EXTRAS|MAKROS|SICHERHEIT.
3. Aktiviere die Option MITTEL.



Makrosicherheit setzen

4. Schließe das Dialogfeld mit OK.
5. Speichere die Arbeitsmappe mit DATEI|SPEICHERN.
6. Schließe Excel, starte es neu und öffne die Arbeitsmappe mit dem Quellcode über DATEI|ÖFFNEN.
7. Excel wird dich nun fragen, ob die Makros deaktiviert werden sollen. Klicke auf die Schaltfläche MAKROS AKTIVIEREN.

Aktivieren der Makros beim Öffnen der Arbeitsmappe

Prozeduren ausführen

Wenn du die Prozedur ausführen willst, geht das am einfachsten, indem du den Cursor in eine der Zeilen der Prozedur setzt.

Anschließend drückst du F5 oder wählst alternativ AUSFÜHREN|SUB|USERFORM AUSFÜHREN. Natürlich kannst du auch die AUSFÜHREN-Schaltfläche der Symbolleiste verwenden.

Das Symbol mit der Sprechblase wird angezeigt, das du mit der Konstanten `vbInformation` festgelegt hast. Hättest du den Parameter ganz weggelassen, was ebenso möglich gewesen wäre, hätte das Dialogfeld kein Symbol, sondern es würde lediglich die Meldung angezeigt.

Code strukturieren

Dass Code in Modulen gespeichert wird, weißt du nun. Aber auch Module haben einen bestimmten Aufbau.

Modulaufbau

Innerhalb eines Moduls kannst du sowohl Prozeduren erstellen als auch Variablen und Konstanten definieren.

Konstanten sind benannte Werte, die sich während des Programmablaufs nicht ändern können.

Üblicherweise werden alle Deklarationen am Anfang des Moduls gemacht; danach folgen die Prozeduren und Funktionen, die im Modul gespeichert werden.

An jeder Stelle im Modul kannst du Kommentare einfügen. Kommentare werden bei der Programmausführung nicht berücksichtigt. Du kannst sie verwenden, um Code zu kommentieren oder den Code eines Moduls zu gliedern.

Kommentare werden mit einem Hochkomma am Zeilenanfang eingeleitet und enden automatisch am Zeilenende.

Du kannst z. B. eine Kommentarzeile am Anfang des Moduls einfügen, mit der du die folgende Prozedur erläuterst:

```
'Gibt eine Meldung aus.
Public Sub Meldung()
    MsgBox "Meldung", vbInformation
End Sub
```

Kommentarzeilen werden in der IDE normalerweise grün dargestellt.

Prozeduren erstellen

Wie oben erläutert sind Prozeduren benannte Codefragmente, die du jederzeit aufrufen kannst. Es gibt prinzipiell zwei Arten von Prozeduren – solche ohne und solche mit Rückgabewert.

Prozeduren mit Rückgabewert werden Funktionen genannt, wohingegen die Prozeduren ohne Rückgabewert Unterprozeduren oder oft einfach Prozeduren genannt werden. Letzteres ist zwar ungenau; du findest diese Bezeichnung aber auch in der Hilfe, weswegen wir sie hier beibehalten werden.

Funktionen sind Prozeduren mit einem Rückgabewert. Einfache Prozeduren, ohne Rückgabewert, werden Prozeduren oder Unterprozeduren genannt.

Wie du Funktionen definierst und verwendest, erfährst du im Abschnitt „Werte aus Funktionen zurückgeben“ auf Seite 14. Nachfolgend geht es ausschließlich um Prozeduren. Eine Prozedur deklarierst du mit dem Schlüsselwort Sub.

Schlüsselwörter sind wichtige Befehle einer Programmiersprache; sie stellen ihren Stammwortschatz da. In objektorientierten Programmiersprachen wie VBA werden diese Schlüsselwörter durch Objekte ergänzt.

Nach dem Schlüsselwort folgt der Name der Prozedur, gefolgt von einem leeren Paar unter Klammern. Dieses gibt an, dass es keine Parameter gibt, die an die Prozedur übergeben werden. Parameter sind Werte, die du an Prozeduren und Funktionen übergeben kannst. Mehr dazu findest du im Abschnitt „Parameter an Prozeduren und Funktionen übergeben“ auf Seite 15.

Nach dem Klammerpaar drückst du Enter, um eine neue Zeile zu erzeugen. Die Entwicklungsumgebung fügt dann automatisch den Prozedurfuß End Sub ein. Zwischen dem Prozedurkopf, also der Zeile, die mit Sub beginnt, und dem Prozedurfuß fügst du die Anweisungen der Prozedur ein.

Der Aufbau einer Prozedur sieht damit folgendermaßen aus:

```
Sub Name_der_Prozedur()
    Anweisungen
End Sub
```

Einfache Anweisungen eingeben

Innerhalb der Prozedur gibst du die Anweisungen ein, die ausgeführt werden sollen, wenn die Prozedur ausgeführt wird. Willst du z. B. eine Prozedur erzeugen, die das aktuelle Datum im Testfenster ausgibt, tust du das in einem normalen Modul:

```
Sub Datum()
    Debug.Print Date
End Sub
```

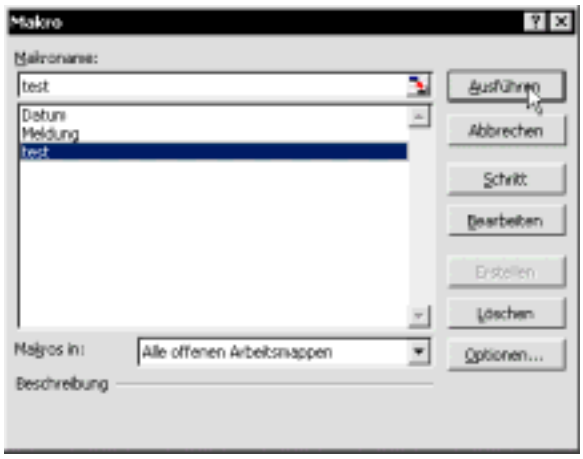
Die Anweisung innerhalb der Prozedur sorgt nun dafür, dass das aktuelle Datum mit `Date` ermittelt und durch Übergabe an die `Debug.Print`-Anweisung im Direktfenster ausgegeben wird.

Prozeduren aufrufen

Du kannst die Prozedur nun direkt mit F5 ausführen oder aber in einer anderen Prozedur aufrufen. Dazu musst du eine Prozedur erstellen, die an der Stelle, wo der Aufruf erfolgen soll, den Namen der aufzurufenden Prozedur nennt – z. B. so:

```
Sub test()
    Datum
End Sub
```

Die Prozedur `Test` könntest du dann z. B. mit F5 aktivieren oder auch aus dem Excel-Fenster durch Aufruf von EXTRAS|MAKRO|MAKROS, Auswahl des Makros per Mausklick auf den Listeneintrag und Klicken auf AUSFÜHREN starten.



Ausführen eines Makros über den Makro-Dialog von Excel

Natürlich ist es nicht gerade sinnvoll, für die Ausgabe des Datums im Testfenster zwei Prozeduren zu erstellen. Du wirst aber gleich merken, dass dies schon dann Sinn macht, wenn Werte an eine Prozedur übergeben werden sollen oder komplexere Berechnungen angestellt werden.

Werte vorübergehend in Variablen speichern

Wenn du Berechnungen durchführen willst, kommst du nicht um die Verwendung von Variablen herum. Du kannst Variablen in einem Modul außerhalb einer Prozedur definieren. Dann handelt es sich um modulweit gültige Variablen, die auf Modulebene definiert sind. Alternativ kannst du Variablen aber auch innerhalb einer Prozedur definieren. Sie sind dann nur innerhalb dieser Prozedur gültig und heißen lokale Variablen.

In jedem Fall definierst du die Variablen mit der `Dim`-Anweisung. Sie hat folgenden Aufbau:

```
Dim Variablenname As Datentyp
```

Der Variablenname darf wie ein Modul- oder Prozedurname nur Buchstaben, Ziffern und Unterstriche enthalten und muss mit einem Buchstaben beginnen. Innerhalb des Gültigkeitsbereichs darf der gleiche Name nur für eine Variable, Prozedur oder ein Modul verwendet werden. Du kannst also keine Variable `Test` auf Modulebene definieren, wenn das Modul schon eine Variable oder eine Prozedur mit dem Namen `Test` enthält.

Der Gültigkeitsbereich einer Variablen definiert, wo und wann du auf den Wert der Variablen zugreifen kannst. Ist die Variable auf Modulebene deklariert, kannst du sie in jeder Prozedur des Moduls nutzen. Wenn sie innerhalb einer Prozedur definiert ist, kannst du sie nur innerhalb der Prozedur verwenden. Der Gültigkeitsbereich ist dann auf die Prozedur begrenzt.

Der Datentyp einer Variablen bestimmt, welche Art von Daten in der Variablen gespeichert werden können. Damit legst du aber auch gleichzeitig den benötigten Speicherplatz fest.

Du kannst z. B. die eben erstellte Prozedur `Datum` dahingehend erweitern, dass ein neues Datum berechnet wird, das ausgehend vom aktuellen Datum zwei Tage in der Zukunft liegt. Das Ergebnis der Berechnung ist dann natürlich wieder ein Datum – also benötigst du zur vorübergehenden Speicherung des Wertes eine Variable des Typs `Date`. Solche Variablen können Datumswerte speichern. Dazu deklarierst du am Anfang der Prozedur die Variable mit:

```
Dim dteDatum As Date
```

Eine Liste der Datentypen findest du auf Seite 65.

Mit der Deklaration wird die Variable dem Compiler und der Entwicklungsumgebung bekannt gemacht. Das System weiß anschließend,

wie die Variable heißt und welche Art Daten sie speichern kann – im Beispiel ein Datum.

Außerdem musst du der Variablen einen Wert zuweisen. Die erste Wertzuweisung wird als Initialisierung bezeichnet. Bei Variablen stellen Deklaration und Initialisierung grundsätzlich zwei Anweisungen dar.

Nach der Deklaration folgt also die Initialisierung. Dazu weist du der Variablen mit dem Zuweisungsoperator „=“ einen Wert zu. Der Wert muss dem Datentyp der Variablen entsprechen.

Da zum aktuellen Datum zwei Tage addiert werden sollen, musst du eine Funktion aufrufen, die den gewünschten Wert zurückgibt. Dafür dient die Funktion `DateAdd`. Sie erwartet drei Parameter, die du getrennt durch Kommata nacheinander angibst. Der erste Parameter mit dem Wert „d“ legt fest, dass der zweite Parameter Tage („d“=days) sind. Der dritte Parameter legt das Datum fest, zu dem die Tage addiert werden sollen. Die Funktion `DateAdd` gibt das Berechnungsergebnis zurück. Dieses wird dann mit Hilfe des Zuweisungsoperators „=“ der Variablen `dteDatum` zugewiesen.

```
Sub Datum()
    Dim dteDatum As Date
    dteDatum = DateAdd("d", 2, Date)
    Debug.Print dteDatum
End Sub
```

Damit das berechnete Datum ausgegeben wird, musst du nun noch die Variable anstelle der `Date`-Anweisung an die `Print`-Anweisung übergeben.

Konstanten definieren und verwenden

Die direkte Nutzung des Wertes 2 in der Funktion `DateAdd` ist nicht immer von Vorteil. Es kann vorkommen, dass du den gleichen Wert in einer Prozedur mehrfach benötigst. Willst du ihn bei Bedarf einfach und schnell ändern, statt ihn manuell an mehreren Stellen zu ersetzen, kannst du Konstanten definieren.

Das kannst du wie bei Variablen ebenfalls auf Modulebene oder innerhalb einer Prozedur machen. Abhängig davon ergibt sich auch der gleiche Gültigkeitsbereich wie bei Variablen.

Im Unterschied zu Variablen deklarierst du Konstanten mit dem Schlüsselwort `Const`. Du kannst wahlweise auch mit `As` einen Datentyp angeben, musst das aber nicht tun, da du bei der Deklaration auch gleich den Wert der Konstanten bestimmst. Damit liegt auch der Datentyp und der verwendete Speicherplatz fest.

Willst du eine Konstante mit dem Wert 2 definieren, kannst du also wahlweise eine der beiden folgenden Anweisungen verwenden:

```
Const bytTage As Byte = 2
Const bytTage = 2
```

Beide definieren die benötigte Konstante, die du einfach anstelle des Wertes 2 an die `DateAdd`-Funktion übergeben kannst:

```
Sub Datum()
    Dim dteDatum As Date
    Const bytTage = 2
    dteDatum = DateAdd("d", _
        bytTage, Date)
    Debug.Print dteDatum
End Sub
```

Überall wo du die Konstante verwendest, wird dann zur Laufzeit, also wenn du den Code ausführst, der Wert der Konstanten verwendet.

Das im Code verwendete Zeichen „ _“ (Leerzeichen + Unterstrich) ist ein Zeilenumbruchzeichen, das dem Compiler mitteilt, dass die Anweisung in der nächsten Zeile fortgesetzt wird. Normalerweise endet eine Anweisung nämlich am Zeilenende. Dieses Zeilenumbruchzeichen besteht grundsätzlich aus einem Leerzeichen gefolgt von einem Unterstrich, und es muss zwingend als letztes Zeichen der Zeile angegeben werden. Mit seiner Hilfe kannst du längere Codezeilen umbrechen, um besser lesbaren Code zu erstellen. In diesem Heft wird der Code vor allem so umgebrochen, dass er in einer Druckzeile Platz hat.

Das Zeilenumbruchzeichen darf allerdings keinesfalls innerhalb von Zeichenketten stehen, die in Anführungszeichen eingefasst werden.

Werte aus Funktionen zurückgeben

In ihrer gegenwärtigen Form ist die Prozedur `Datum` natürlich noch schlecht einsetzbar. Statt das berechnete Datum im Direktfenster auszugeben, wäre es natürlich besser, dieses Datum

aus der Prozedur zurückzugeben, um es so in anderen Ausdrücken verwenden zu können. Du könntest das Ergebnis dann immer noch an die `Print`-Anweisung übergeben – du könntest es aber genauso gut anders verwenden.

Soll das funktionieren, musst du aus der Prozedur eine Funktion machen, da nur Funktionen einen Wert zurückgeben können.

Funktionen werden mit dem Schlüsselwort `Function` definiert und enden mit `End Function`. Nach dem runden Klammernpaar folgt das Schlüsselwort `As` – und darauf dann der Datentyp des Rückgabewertes. Innerhalb der Funktion muss eine Anweisung stehen, die dem Funktionsnamen den Rückgabewert zuweist. Der Aufbau einer Funktion lautet:

```
Function Funktionsname() _
    As Datentyp
    ...
    Funktionsname = Wert
End Function
```

Willst du aus der bisherigen Prozedur eine Funktion machen, musst du die nachfolgend fett hervorgehobenen Änderungen durchführen. Den Namen kannst du beibehalten – es sei denn, du möchtest die Prozedur bewahren und eine zusätzliche Funktion erstellen.

```
Function Termin() As Date
    Dim dteDatum As Date
    Const bytTage = 2
    dteDatum = DateAdd("d", _
        bytTage, Date)
    Termin = dteDatum
End Function
```

Willst du eine Funktion aufrufen, geht das genauso wie bei Prozeduren. Du kannst die Funktion z. B. mit der Anweisung `Termin` aufrufen.

In diesem Falle wird die Funktion zwar aufgerufen und auch der Rückgabewert berechnet – aber er wird nicht zurückgegeben. Das liegt daran, dass du keine Variable angegeben hast, der der Rückgabewert zugewiesen wird. Um das zu erreichen, musst du eine Variable deklarieren und ihr den Rückgabewert der Funktion zuweisen:

```
Sub test()
    Dim dteTermin As Date
    dteTermin = Termin()
End Sub
```

Bei der Zuweisung des Rückgabewertes an eine Variable musst du zwingend das leere Klammerpaar nach dem Funktionsnamen angeben.

Statt den Rückgabewert einer Variable zuzuweisen kannst du ihn aber auch unmittelbar an eine Prozedur oder Anweisung übergeben – etwa so:

```
Debug.Print Termin()
```

Parameter an Prozeduren und Funktionen übergeben

Natürlich ist es nicht immer wünschenswert, dass zum aktuellen Datum genau zwei Tage addiert werden. Soll die Funktion (analog gilt das auch für Prozeduren) flexibler einsetzbar sein, kannst du die Tage, die zum Datum addiert werden sollen, auch als Parameter übergeben.

Parameter definierst du, indem du sie innerhalb des runden Klammerpaars im Funktionskopf angibst. Jeden Parameter definierst du nach folgendem Schema:

```
ParamName As Datentyp
```

Willst du mehrere Parameter definieren, führst du sie nacheinander auf und trennst sie durch Kommata voneinander ab.

Im folgenden Code werden zwei Parameter definiert. Der erste hat den Typ `Integer` und kann damit ganze Zahlen von -32768 bis +32767 speichern. Der zweite Parameter stellt das Datum dar, zu dem die Tage addiert werden sollen.

```
Function Termin(intTagDiff As _
    Integer, dteDatum As Date) _
    As Date
...
End Function
```

Innerhalb der Funktion kannst du die Parameter wie Variablen verwenden – nur musst du sie nicht mehr deklarieren und initialisieren. Da der Parameter `dteDatum` vorher als Variable deklariert war, musst du nur die beiden Anweisungen zur Deklaration und Initialisierung löschen.

```
Function Termin(intTagDiff As _
    Integer, dteDatum As Date) _
    As Date
    Dim dteDatum As Date
    Const bytTage As Byte = 2
    Const bytTage = 2
    dteDatum = DateAdd("d", _
```

```
        intTagDiff, Date)
    Termin = dteDatum
End Function
```

Hast du Parameter definiert, so musst du für diese auch Werte an die Funktion übergeben, wenn du sie aufrufst. Im einfachsten Fall übergibst du die Werte einfach der Reihe nach und trennst sie durch Kommata. Dabei musst du Zeichenketten (Datentyp `String`) in Anführungszeichen setzen und Datumswerte mit dem Zeichen „#“ einklammern. Datumsangaben kannst du also entweder als Variable vom Typ `Date` oder mit dem Schema `#MM/TT/JJJJ#` übergeben. In der folgenden Anweisung wird als Datum der 27.09.2004 angegeben:

```
dteTermin = Termin(2, #9/27/2004#)
```

Alternativ kannst du die Parameter in beliebiger Reihenfolge übergeben, musst dann aber den Parameternamen angeben. Die Übergabe erfolgt dann nach folgendem Schema:

```
ParamName:=Wert
```

Willst du mehrere Parameter übergeben, trennst du sie einfach durch ein Komma:

```
dteTermin = _
    Termin(dteDatum:=#9/27/2004#, _
    intTagDiff:=2)
```

Dies Form des Aufrufs sieht zwar länger aus, doch hat sie durchaus Vorteile. Zum einen siehst du, wenn viele Parameter übergeben werden, auf den ersten Blick, welcher Wert an welchen Parameter übergeben wird; zum anderen kannst du bei optionalen Parametern die lästigen leeren Parameterwerte weglassen und dich auf den vielleicht einzigen übergebenen Wert beschränken.

Optionale Parameter sind Parameter, die an eine Prozedur oder Funktion übergeben werden können, aber nicht müssen.

Die Parameterübergabe an eine Prozedur funktioniert fast genauso – nur lässt du die runden Klammern weg. Wäre die Funktion Termin eine Prozedur oder möchtest du den Rückgabewert nicht verwenden, könnte der Aufruf wie folgt aussehen:

```
Termin 2, #9/27/2004#
```

Prozeduren und Funktionen mit Parametern kannst du nicht mit F5 ausführen. Du benötigst grundsätzlich eine parameterlose Prozedur, die wiederum die Prozedur oder Funktion bei Übergabe der Parameter aufruft.

Optionale Parameter definieren

Die Deklaration der beiden Parameter war natürlich eigentlich ein kleiner Rückschritt, weil nun fast genauso viele Werte an die Funktion übergeben werden müssen wie an die Funktion `DateAdd`.

Du kannst die Funktion (analog geht das auch bei Prozeduren) aber so erweitern, dass die Parameterwerte übergeben werden können, aber nicht müssen. Für den Fall, dass die Werte nicht übergeben werden, musst du dann einen Standardwert bestimmen.

Optionale Parameter definierst du, indem du das Schlüsselwort `optional` vor den Parameter setzt.

Generell gilt, dass alle nach einem optionalen Parameter folgenden Parameter ebenfalls optional sein müssen. Definierst du also den ersten Parameter als optional, muss es der zweite auch sein. Alternativ könntest du den optionalen Parameter an das Ende der Parameterliste setzen. Die Parameter vor dem optionalen kannst du dann ohne `optional` definieren. Mit folgendem Funktionskopf legst du beide Parameter als optional fest.

```
Function Termin(Optional _
    intTagDiff As Integer, _
    Optional dteDatum As Date) _
    As Date
```

Das Problem ist, dass du so noch keine Standardwerte festgelegt hast, für den Fall, dass die Parameter nicht übergeben werden. Dazu gibt es zwei Möglichkeiten. Die eine: Du weist dem Parameter im Prozedurkopf einen konstanten Wert zu, z. B. je nach Datentyp des Parameters eine Zahl oder einen Text. Alternativ deklarierst du den Parameter mit dem Datentyp `Variant` und prüfst erst in der Funktion, ob der Wert übergeben wurde. Dazu gibt es eine spezielle Funktion, `IsMissing`.

Der Datentyp `Variant` ist ein allgemeiner Datentyp, der alle Arten von Daten speichern kann. Damit benötigt er aber auch sehr viel Speicherplatz. Also solltest du diesen Datentyp nur verwenden, wenn es unumgänglich ist, wie in diesem Beispiel.

Willst du einem Parameter einen konstanten Standardwert zuweisen, fügst du den Wert einfach an den Datentyp an. Hier wird z.B. der Wert 2 als Standardwert für den ersten Parameter definiert. Der zweite Parameter soll als Standardwert das aktuelle Datum bekommen, das von der Funktion `Date` zurückgegeben wird. Da einem Parameter nur konstante Werte zugewiesen werden können, nicht aber Funktionen, musst du den Standardwert über die `IsMissing`-Funktion festlegen. Dazu löschst du einfach die `As`-Anweisung mit dem Datentyp und legst damit den Datentyp `Variant` für den Parameter fest.

Alternativ könntest du auch den Datentyp `Date` explizit durch `Variant` ersetzen.

```
Function Termin(Optional _
    intTagDiff As Integer = 2, _
    Optional dteDatum As Date) _
    As Date
    If IsMissing(dteDatum) Then
        dteDatum = Date
    End If
    dteDatum = DateAdd("d", _
        intTagDiff, Date)
    Termin = dteDatum
End Function
```

Innerhalb der Funktion musst du nun mit der `IsMissing`-Funktion prüfen, ob der zweite Parameter übergeben wurde. Dazu ist eine `if`-Anweisung notwendig. Sie prüft, ob der Ausdruck `IsMissing(dteDatum)` wahr ist; das ist dann der Fall, wenn der Parameter nicht angegeben wurde. Nur in diesem Falle wird die Anweisung `dteDatum=Date` ausgeführt und dem Parameter das aktuelle Datum zugewiesen.

Mehr zu `If`-Anweisung findest du weiter unten im Abschnitt „Verzweigungen und Schleifen“ auf Seite 23.

Optionale Parameter zeichnen sich dadurch aus, dass sie nicht angegeben werden müssen. Das ist dann natürlich beim Aufruf der Prozedur oder Funktion wichtig. Gibst du die Parameter in der definieren Reihenfolge an, ohne die Parameternamen zu nennen, lässt du optionale Parameter einfach weg, indem du schlicht die Kommata nach dem vorherigen und vor dem nächsten Parameter angibst, ohne Werte zwischen diese Kommata zu setzen. Gibst du die Parameternamen an, nennst du nur die Parameter, an die du Werte übergeben möchtest. Möchtest du z.B. an die Funktion `Termin` nur den zweiten Parameter übergeben, kannst du die folgenden Aufrufe verwenden:

```
Debug.Print Termin( ,#01/17/2004#)
Debug.Print Termin( dteDatum:= _
    #01/17/2004#)
```

Alternativ könntest du nun natürlich auch beide Parameter weglassen:

```
Debug.Print Termin()
```

Vielleicht ist dir bei Eingabe des Funktionsaufrufs aufgefallen, dass in der Parameterliste nun beide Parameter von eckigen Klammern eingefasst werden. So zeigt die Programmierhilfe optionale Parameter an. Der mit dem Zuweisungsoperator im Prozedurkopf definierte Standardwert wird ebenfalls angezeigt.

Darstellung der optionalen Parameter in der Parameterliste

Besondere Prozeduren

Excel unterstützt zwei besondere Prozeduren. Mit einer Prozedur mit dem Namen `Auto_Open` und einer leeren Parameterliste kannst du Code definieren, der beim Öffnen der Arbeitsmappe ausgeführt wird. Entsprechend werden Anweisungen in einer Prozedur mit dem Namen `Auto_Close` ausgeführt, wenn die Arbeitsmappe geschlossen wird.

```
Sub Auto_Open()
    MsgBox "Dieser Code wird " & _
        "beim Öffnen ausgeführt!"
End Sub
```