HANSER

Leseprobe

zu

„Excel programmieren“

von Michael Kofler und Ralf Nebelo

ISBN (Buch): 978-3-446-43866-8
ISBN (E-Book): 978-3-446-43912-2

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/978-3-446-43866-8
sowie im Buchhandel

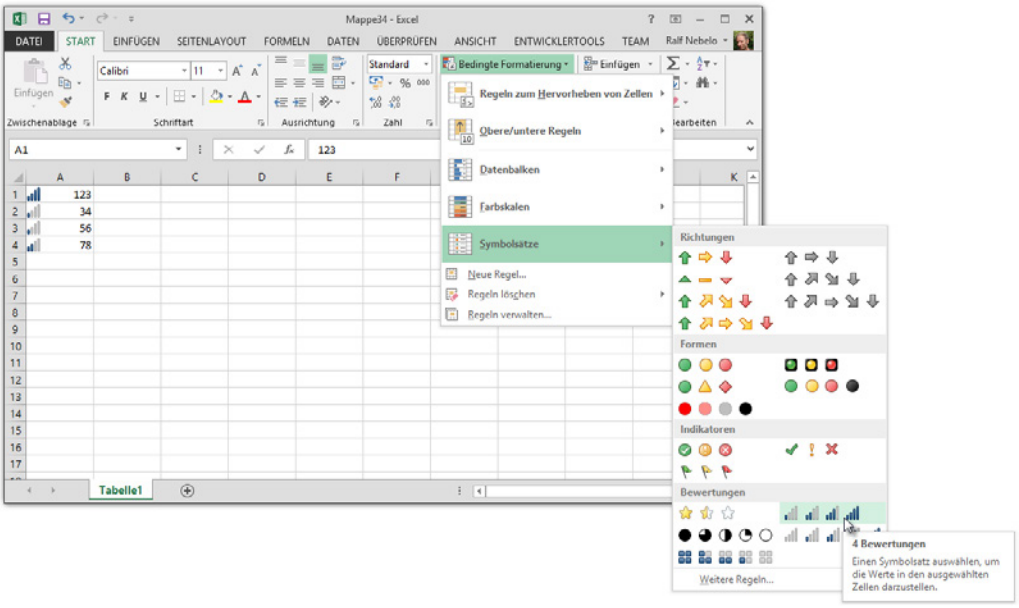

BILD 10.10 Die Zelldiagramme der bedingten Formatierung setzen für die grafische Darstellung von Zellwerten unter anderem Symbolsätze ein.

10.5.1 Programmierung von Datenbalkendiagrammen

Das programmierte Anlegen eines Datenbalkendiagramms beginnt mit dem Aufruf der *AddDatabar*-Methode. Die fügt einem beliebigen Arbeitsblattbereich (*Range*-Objekt) oder der aktuellen Auswahl (*Selection*-Objekt) eine neue bedingte Formatierung des Typs „Datenbalken" hinzu. Den Verweis darauf sollten Sie in einer Objektvariablen vom Datentyp *Databar* speichern:

```
Dim objBalkenFormat As Databar
Set objBalkenFormat = Selection.FormatConditions.AddDatabar
```

Über die Eigenschaften und Methoden der *Databar*-Variablen lassen sich nun die Details des Datenbalkendiagramms festlegen. Über die *ShowValue*-Methode beispielsweise können Sie bestimmen, ob innerhalb der markierten Zellen Zahlenwerte angezeigt werden sollen (*True*) oder nur Diagrammelemente (*False*):

```
objBalkenFormat.ShowValue = True
```

Für die Darstellung des Diagramms ist es wichtig, woher der kürzeste und der längste Balken ihre Werte beziehen. Standardmäßig sind das der kleinste und der größte Zellwert innerhalb des jeweiligen Arbeitsblattbereichs. Dieser Bezug lässt sich aber mithilfe der *Modify*-Methode der untergeordneten Objekte *MinPoint* und *MaxPoint* verändern. So könnte man den kürzesten und längsten Balken etwa die Werte 100 und 300 fest zuordnen:

```
objBalkenFormat.MinPoint.Modify xlConditionValueNumber, 100
objBalkenFormat.MaxPoint.Modify xlConditionValueNumber, 300
```

Die *PercentMax*-Eigenschaft bestimmt, wie viel Prozent der Zellbreite der Diagrammbalken maximal einnehmen darf. Im Normalfall liegt der Wert bei 100 (volle Zellbreite), lässt sich wie folgt aber beispielsweise auf die Hälfte reduzieren:

```
objBalkenFormat.PercentMax = 50
```

Über das Unterobjekt *BarColor* können Sie die Farbe der Diagrammbalken wählen, indem Sie seiner *Color*-Eigenschaft einen entsprechenden RGB-Wert (im Beispiel ein helles Grün) zuweisen:

```
objBalkenFormat.BarColor.Color = RGB(0, 200, 0)
```

Hinweis

Die Zuweisung von Farbwerten geschieht in der Regel mithilfe der VBA-Funktion *RGB*. Die besitzt drei numerische Argumente, über die Sie die gewünschten Intensitäten der Grundfarben Rot, Grün und Blau festlegen und damit beliebige Farben mischen können. Die möglichen Werte liegen zwischen 0 (Farbe fehlt ganz) und 255 (Farbe hat maximale Helligkeit).

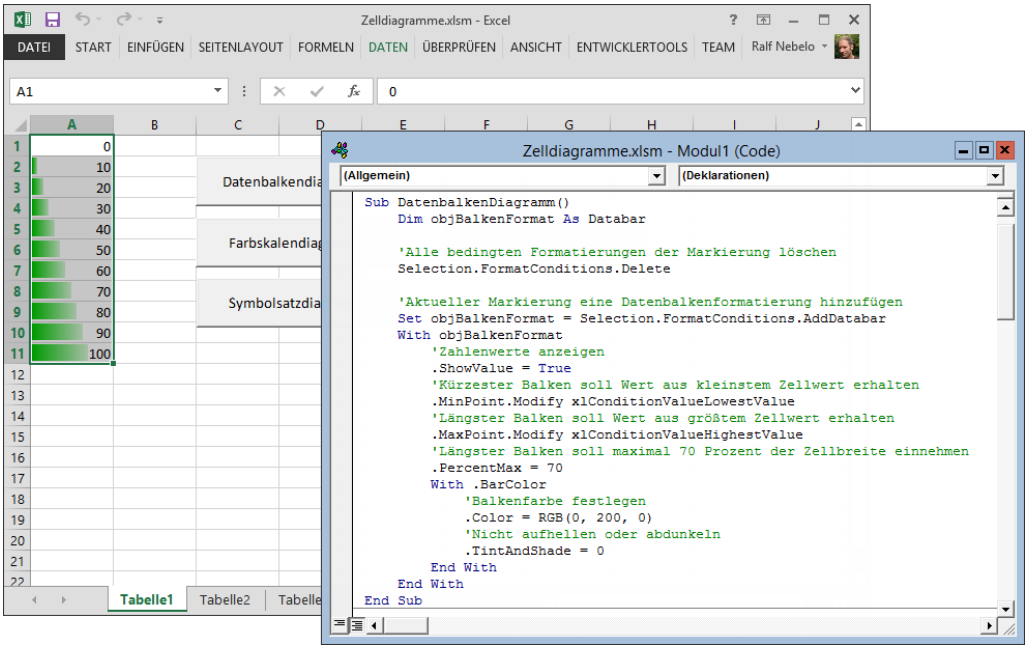

BILD 10.11 Ein Datenbalkendiagramm und das VBA-Makro, dem es seine Existenz verdankt

10.5.2 Programmierung von Farbskalendiagrammen

Beim programmierten Erstellen eines Farbbalkendiagramms kommt zunächst die *AddColorScale*-Methode zum Einsatz. Die fügt den in der *FormatConditions*-Auflistung gesammelten bedingten Formatierungen des Arbeitsblattbereichs eine neue Formatierung vom Typ „Farbskala“ hinzu. Das numerische Argument der Methode bestimmt, wie viele Farbbereiche die Skala enthalten soll. Mögliche Werte sind 2 und 3. Den Verweis auf das neu angelegte Formatierungsobjekt nimmt eine Variable vom Datentyp *ColorScale* auf:

```
Dim objFarbskalenFormat As ColorScale
Set objFarbskalenFormat = _
Selection.FormatConditions.AddColorScale(3)
```

Über die *ColorScaleCriteria*-Auflistung und eine Indexzahl ist der Zugriff auf jeden einzelnen Farbbereich möglich, um dessen Eigenschaften zu bestimmen.

- So kann man etwa die gewünschte Startfarbe des Farbbereichs festlegen, indem man seiner *FormatColor.Color*-Eigenschaft einen passenden RGB-Wert zuweist.
- Die *Type*-Eigenschaft des *ColorScale*-Objekts legt fest, für welchen Wertebereich der Farbbereich zuständig ist.

Das Beispiel

```
With objFarbskalenFormat.ColorScaleCriteria(1)
    .Type = xlConditionValueLowestValue
    .FormatColor.Color = RGB(150, 0, 0)
End With
```

regelt, dass der erste der drei Farbbereiche (*ColorScaleCriteria(1)*) für die Darstellung der niedrigsten Werte (*xlConditionValueLowestValue*) zuständig ist und dafür Abstufungen der Farbe Rot (*RGB(150, 0, 0)*) verwendet.

Sollen die mittleren Zellwerte in die Zuständigkeit des zweiten Farbbereichs (*ColorScaleCriteria(2)*) fallen, muss man seiner *Type*-Eigenschaft den Wert *xlConditionValuePercentile* zuordnen und zusätzlich seine *Value*-Eigenschaft auf 50 setzen. Das Beispiel weist dem Farbbereich darüber hinaus ein mittleres Grün (*RGB(0, 150, 0)*) als Startfarbe zu:

```
With objFarbskalenFormat.ColorScaleCriteria(2)
    .Type = xlConditionValuePercentile
    .Value = 50
    .FormatColor.Color = RGB(0, 150, 0)
End With
```

Beim dritten Farbbereich (*ColorScaleCriteria(3)*) genügen dann wieder zwei Eigenschaftszuweisungen. Das Beispiel

```
With objFarbskalenFormat.ColorScaleCriteria(3)
    .Type = xlConditionValueHighestValue
```

```
    .FormatColor.Color = RGB(0, 0, 150)
End With
```

macht diesen für die Darstellung der höchsten Zellwerte verantwortlich (*.Type = xlConditionValueHighestValue*), wobei ein mittleres Blau (*RGB(0, 0, 150)*) als Startfarbe zum Einsatz kommt.

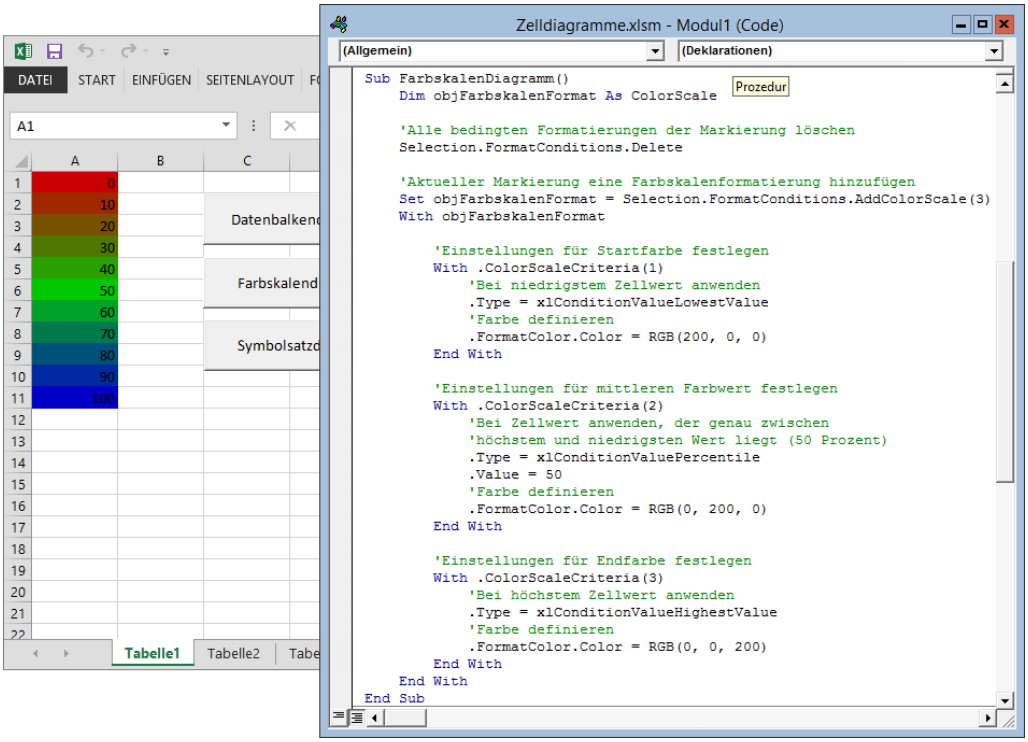

BILD 10.12 Farbskalendiagramme kennzeichnen die unterschiedlichen Zellwerte durch Farben und Farbabstufungen.

10.5.3 Programmierung von Symbolsatzdiagrammen

Die *AddIconSetCondition*-Methode fügt der Liste der bedingten Formatierungen eine neue *Symbolsatz*-Formatierung hinzu, die in einer Variablen vom Typ *IconSetCondition* gespeichert wird:

```
Dim objSymsatz As IconSetCondition
Set objSymsatz = Selection.FormatConditions.AddIconSetCondition
```

Die Objektvariable stellt Ihnen mehrere Eigenschaften zur Verfügung, über die Sie das Diagramm nach Ihren Wünschen gestalten können.

- Die *IconSet*-Eigenschaft bestimmt den Symbolsatz, der von der *IconSets*-Auflistung der Arbeitsmappe bereitgestellt wird. Da diese Auflistung standardmäßig 20 Elemente umfasst, erfolgt die Angabe des gewünschten Symbolsatzes über eine Zahl zwischen 1 und 20. Alternativ können Sie auch eine der folgenden Konstanten verwenden:

Konstante	Wert	Symbolsatz
xl3Arrows	1	3 Pfeile (farbig)
xl3ArrowsGray	2	3 Pfeile (grau)
xl3Flags	3	3 Kennzeichen
xl3Signs	6	3 Zeichen
xl3Stars	18	3 Sterne
xl3Symbols	7	3 Symbole (mit Kreis)
xl3Symbols2	8	3 Symbole (ohne Kreis)
xl3TrafficLights1	4	3 Ampeln (ohne Rand)
xl3TrafficLights2	5	3 Ampeln (mit Rand)
xl3Triangles	19	3 Dreiecke
xl4Arrows	9	4 Pfeile (farbig)
xl4ArrowsGray	10	4 Pfeile (grau)
xl4CRV	12	4 Bewertungen
xl4RedToBlack	11	Rot/Schwarz
xl4TrafficLights	13	4 Ampeln
xl5Arrows	14	5 Pfeile (farbig)
xl5ArrowsGray	15	5 Pfeile (grau)
xl5Boxes	20	5 Kästchen
xl5CRV	16	5 Bewertungen
xl5Quarters	17	5 Viertel

- Mit *ReverseOrder* legen Sie fest, ob die Zuordnung von Zellwerten und Symbolen in der normalen (*False*) oder umgekehrten Reihenfolge (*True*) erfolgen soll.
- Die Eigenschaft *ShowIconOnly* regelt, ob innerhalb der Zellen nur noch die Symbole angezeigt werden (*True*) oder Symbole *und* Zellwerte (*False*).

Das folgende Beispiel weist dem Diagramm den Symbolsatz „3 Ampeln (mit Rand)“ zu, kehrt die Zuordnung von Zellwerten und Symbolen um, lässt aber beide innerhalb der Zellen anzeigen:

```
With objSymsatz
    .IconSet = ActiveWorkbook.IconSets(xl3TrafficLights2)
    .ReverseOrder = True
    .ShowIconOnly = False
End With
```

Das Diagramm stellt für jedes Symbol des gewählten Symbolsatzes ein *IconCriteria*-Element zur Verfügung. Das besitzt die folgenden drei Eigenschaften, mit denen Sie festlegen können, ab welchem Wertebereich das jeweilige Symbol zum Einsatz kommt:

- Die *Type*-Eigenschaft bestimmt, ob die Wertbestimmung absolut (*xlConditionValueNumber*), prozentual (*xlConditionValuePercent*) oder als Ergebnis einer Rechenformel (*xlConditionValueFormula*) erfolgen soll.
- Im Falle einer prozentualen oder absoluten Bestimmung nennt die *Value*-Eigenschaft den gewünschten Prozent- oder Absolutwert.
- Die *Operator*-Eigenschaft definiert den Vergleichsoperator, der in Bezug auf den angegebenen *Value*-Wert zum Einsatz kommt. Die Auswahl erfolgt mithilfe einer der folgenden Konstanten:

Konstante	Wert	Bedeutung
xlEqual	3	gleich
xlGreater	5	größer als
xlGreaterEqual	7	größer oder gleich
xlLess	6	kleiner als
xlLessEqual	8	kleiner oder gleich
xlNotEqual	4	ungleich

Da Excel den Wertebereich des ersten Symbols automatisch festlegt (auf die niedrigsten Zellwerte nämlich), müssen Sie nur noch für die restlichen Symbole die entsprechenden Einstellungen vornehmen. Das folgende Beispiel bestimmt, dass die Zuständigkeit des zweiten Symbols (*IconCriteria(2)*) bei einem Wert beginnt, der 33 oder mehr Prozent des höchsten Zellwerts beträgt:

```
With objSymsatz.IconCriteria(2)
    .Type = xlConditionValuePercent
    .Value = 33
    .Operator = xlGreaterEqual
End With
```

Das Beispiel für das dritte Symbol (*IconCriteria(3)*) lässt dessen Verwendung bei 67 Prozent des höchsten Zellwerts beginnen:

```
With objSymsatz.IconCriteria(3)
    .Type = xlConditionValuePercent
    .Value = 67
    .Operator = xlGreaterEqual
End With
```

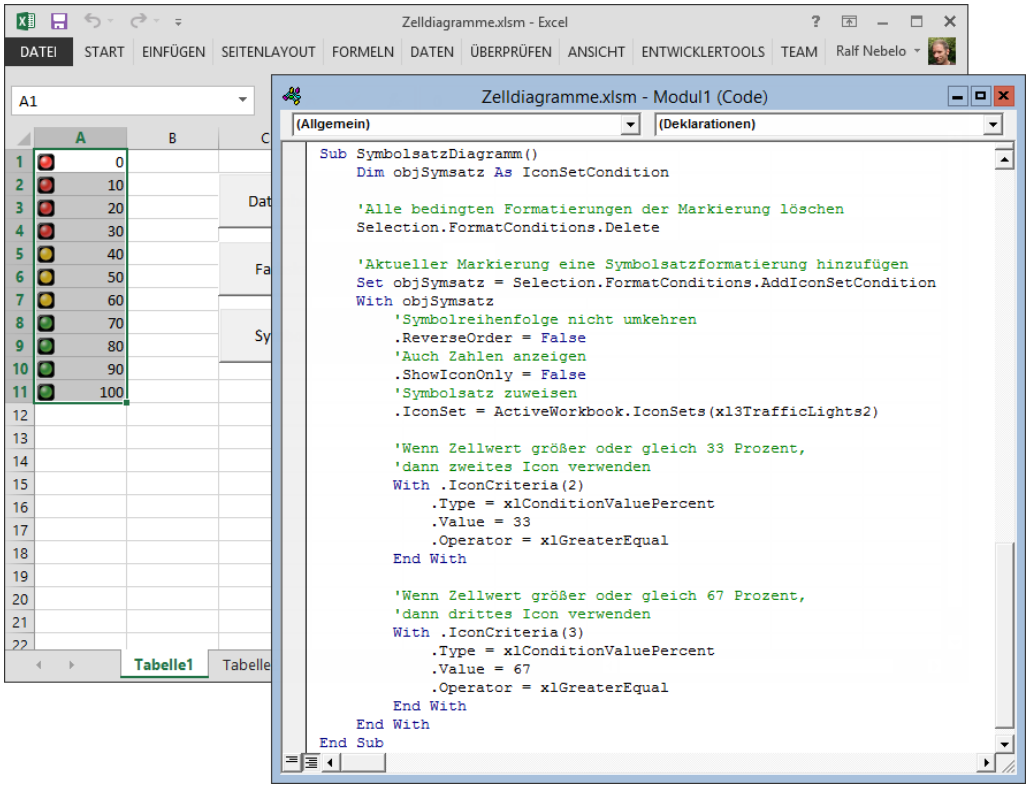

BILD 10.13 Symbolsatzdiagramme erlauben eine besonders schnelle Beurteilung von Zellwerten.

10.5.4 Syntaxzusammenfassung

VBA-Elemente für die Programmierung von Zelldiagrammen	
AddColorScale	erstellt ein Farbskalendiagramm
AddDatabar	erstellt ein Datenbalkendiagramm
AddIconSetCondition	erstellt ein Symbolsatzdiagramm
BarColor	verweist auf die Farbe eines Datenbalkendiagramms
ColorScale	erlaubt Objekterstellung für Farbskalendiagramm
ColorScaleCriteria	verweist auf die Farbbereiche eines Farbskalendiagramms
DataBar	erlaubt Objekterstellung für Datenbalkendiagramm
FormatColor	verweist auf die Farbe eines Farbbereichs in einem Farbskalendiagramm
IconCriteria	verweist auf ein Symbol in einem Symbolsatzdiagramm
IconSet	bestimmt den Symbolsatz eines Symbolsatzdiagramms
IconSetCondition	erlaubt Objekterstellung für Symbolsatzdiagramm
MaxPoint	verweist auf den höchsten Wert in einem Datenbalkendiagramm

VBA-Elemente für die Programmierung von Zelldiagrammen	
MinPoint	verweist auf den niedrigsten Wert in einem Datenbalkendiagramm
Modify	legt den höchsten oder niedrigsten Wert in einem Datenbalkendiagramm fest
Operator	bestimmt den Vergleichsoperator für ein Symbol in einem Symbolsatzdiagramm
PercentMax	bestimmt, wie viel Prozent der Zellbreite das Datenbalkendiagramm maximal einnehmen darf
ReverseOrder	legt die Reihenfolge der Symbole in einem Symbolsatzdiagramm fest
ShowIconOnly	bestimmt, ob die Zellen eines Symbolsatzdiagramms nur Symbole anzeigen sollen
ShowValue	bestimmt, ob Zahlenwerte in Datenbalkendiagrammen angezeigt werden
Type	legt fest, für welchen Wertebereich der Farbbereich eines Farbskalendiagramms zuständig ist
Value	bestimmt den Wert eines Farbbereichs in einem Farbskalendiagramm beziehungsweise den Prozent- oder Absolutwert für ein Symbol in einem Symbolsatzdiagramm

10.6 Sparklines-Diagramme

Die mit Excel 2010 eingeführten Sparklines lassen sich über die Registerkarte EINFÜGEN in das Arbeitsblatt integrieren. Bei diesen „Funkenlinien“ (wie die wörtliche, aber inoffizielle Übersetzung lauten würde) handelt es sich um kleine Linien- oder Balkendiagramme, die im Unterschied zu den „richtigen“ Excel-Diagrammen vollständig in eine Zelle passen. Dadurch lassen sich die Minidiagramme in unmittelbarer Nähe zu ihren Quelldaten platzieren, was sie zum optimalen Mittel für die datennahe Visualisierung von Trends macht – von Wirtschaftszyklen etwa, Ausgabenentwicklungen oder saisonalen Auf- und Abschwüngen.

Aufgrund der geringen Größe sind die Gestaltungsmöglichkeiten von Sparklines-Diagrammen natürlich deutlich eingeschränkt. Der Anwender hat die Wahl zwischen den drei Typen „Linie“, „Säule“ und „Gewinn/Verlust“. Alle Diagrammtypen bieten die Möglichkeit, Maximal- und Minimalwerte sowie negative Zahlen farbig hervorzuheben. Dabei lässt sich die Lage des Nullpunkts durch eine horizontale Achse kennzeichnen. Beim Diagrammtyp *Linie* besteht darüber hinaus die exklusive Möglichkeit, farbige Markierungen für alle Datenpunkte sichtbar zu machen.

Hinweis

Den Code dieses Abschnitts finden Sie in der Datei *SparkLines.xlsm* im Unterordner 10 der Beispieldateien.

15.6.7 Syntaxzusammenfassung

sh steht für ein Tabellen- oder Diagrammblatt, *oleob* für ein OLE-Objekt.

Programmstart und -steuerung	
id= Shell(„datname“)	fremdes Programm starten
AppActivate „fenstertitel“	bereits laufendes Programm
AppActivate id	aktivieren
Application.ActivateMicrosoftApp xlXy	MS-Programm starten/aktivieren
SendKeys „..“	simuliert Tastatureingabe

OLE, ActiveX-Automation	
sh.OLEObjects(..)	Zugriff auf OLE-Objekte
sh.OLEObjects.Add ...	neues OLE-Objekt erzeugen
sh.Pictures.Paste link:=True	OLE-Objekt aus Zwischenablage einfügen
oleob.Select	wählt Objekt aus (normaler Mausklick)
oleob.Activate	aktiviert Objekt (Doppelklick)
oleob.Verb xlOpen/xlPrimary	führt OLE-Kommando aus
oleob.Update	aktualisiert verknüpftes OLE-Objekt
oleob.Delete	löscht OLE-Objekt
oleob.Object	Verweis für ActiveX-Automation
obj = GetObject(„“, „ole-bezeichn“)	Verweis für ActiveX-Automation

15.7 64-Bit-Programmierung

Seit der Versionsnummer 2010 wird Excel sowohl in einer 32- als auch einer 64-Bit-Version angeboten. Letztere bietet den Vorteil, den gesamten Arbeitsspeicher eines 64-Bit-Windows-Systems nutzen zu können. Davon profitiert das Programm insbesondere beim Umgang mit sehr großen Arbeitsmappen, die mehr als 2 GByte RAM für sich beanspruchen dürfen. Ein weiterer 64-Bit-Vorteil ist die sogenannte hardware-gestützte Datenausführungsverhinderung. Die soll Sicherheitslücken im Zusammenhang mit Pufferüberlaufen schließen und damit die Angriffsfläche für Viren und Würmer deutlich verringern.

15.7.1 Kompatibilitätsprobleme

Den genannten Vorteilen stehen allerdings diverse Nachteile gegenüber, die insbesondere die Kompatibilität mit vorhandenen Makros, Add-ins, Datenbanken und sonstigen Excel-Erweiterungen betreffen. So verwendet die 64-Bit-Version von Excel beispielsweise eine ebenso breite Variante des Windows-eigenen *Graphics Device Interface* (GDI), um ihre Diagramme und sonstigen Hochglanzgrafiken zu rendern. Das 64-Bit-GDI unterstützt aber keine MMX-Befehle mehr, die eine besonders schnelle, weil parallele Verarbeitung von Grafik- und Videodaten auf Intel-Prozessoren ermöglichen. Das stellt allen Excel-Erweiterungen mit MMX-gestützten Multimedia-Ambitionen den Stuhl vor die 64-Bit-Tür. Add-ins dieser Art dürfte es allerdings nicht allzuviele geben.

Da wiegt der Ausschluss aller kompilierten Datenbankdateien (*.mde oder *.accde), die je mit einer 32-Bit-Ausgabe von Microsoft Access erstellt wurden, deutlich schwerer. Eine Neukompilierung mit der jüngsten 64-Bit-Variante des Datenbankprogramms kann dieses Problem zwar lösen, erfordert allerdings Zugriff auf die originären ACCDB- oder MDB-Dateien. Die jedoch rücken die Entwickler nur selten heraus, da die verwendeten Programmcodes darin für jedermann einsehbar sind.

Problemfall ActiveX

Noch schwerwiegender dürfte die Verweigerungshaltung von Excel 64 Bit in Bezug auf *ActiveX-Steuerelemente* und *COM-Add-Ins* (siehe Abschnitt 15.1) sein. Dabei handelt es sich durchweg um 32-Bit-Binärdateien, die ein 64-Bit-Prozess grundsätzlich nicht laden kann. Und das ist ein wirkliches Problem, da nahezu jede programmierte Lösung, die die funktionalen Grenzen von Excel wirksam erweitert, COM- und ActiveX-Elemente verwendet: als Userform-Control (Steuerelement) für besondere Aufgaben, als Funktionsbibliothek oder Fernsteuerung für beliebige (COM-fähige) Anwendungen beispielsweise.

Zwar lässt sich auch dieses Problem grundsätzlich durch Neukompilierung beseitigen. Dazu braucht es allerdings einen geeigneten 64-Bit-Compiler, sämtliche Quellcodes sowie ein nicht unerhebliches Know-how. Als Endanwender ohne Programmiererfahrung wird man daher in der Regel warten müssen, bis der Software-Hersteller eine 64-Bit-Version seines ActiveX-Controls beziehungsweise COM-Add-Ins herausgibt.

Problemfall Windows-API

Wenn ein Excel-Entwickler die VBA-Grenzen sprengen will, verwendet er ebenfalls sehr häufig das *Application Programming Interface* (API) von Windows, das seine zahllosen Funktionen in Form von Dynamic Link Libraries (siehe Abschnitt 15.5) zur Verfügung stellt. Für die Verwaltung von Fenster-Handles und Adresszeigern verwenden API-Funktionen standardmäßig den 32-Bit-Datentyp *Long* – was in einem 64 Bit breiten Adressraum schwere Komplikationen bis hin zum Programmabsturz verursachen kann.

Probleme dieser Art kann der Entwickler allerdings selbst lösen – mit den einschlägigen Werkzeugen der neuen VBA-Version 7.0, die wir Ihnen im Folgenden anhand eines praktischen Beispiels vorstellen möchten.

Hinweis

Microsoft bietet ein nützliches Tool an, mit dem Sie die 64-Bit-Verträglichkeit vorhandener Excel-Erweiterungen überprüfen können. Sie finden den *Microsoft Office Code Compatibility Inspector* unter der folgenden Adresse *[Link 35]*:

http://www.microsoft.com/downloads/details.aspx?FamilyID=23C8A7F6-88B3-48EF-9710-9742340562C0

15.7.2 Ein problematisches (32-Bit-)Beispiel

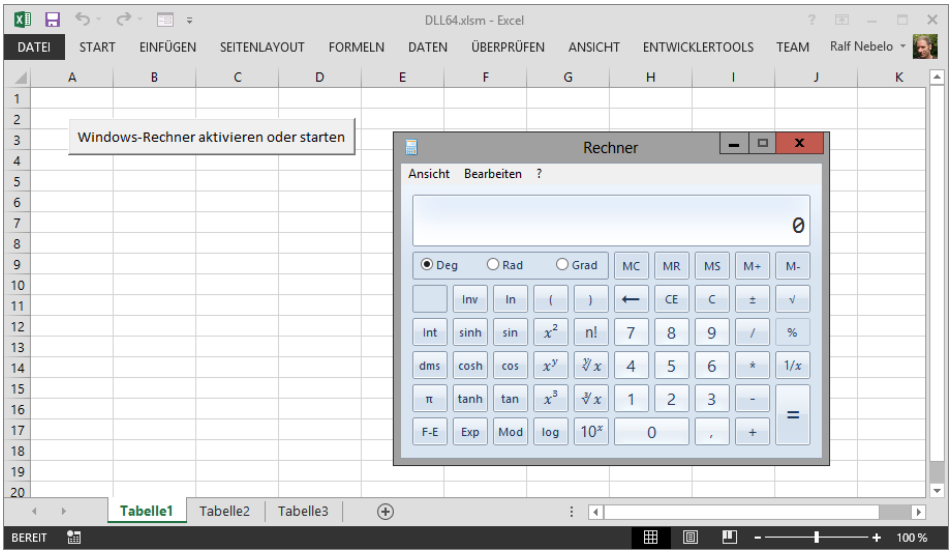

BILD 15.13 Das Beispielprogramm nutzt diverse API-Funktionen, die den Windows-Rechner je nach Zustand starten oder aktivieren.

Wenn ein VBA-Makro eine externe Anwendung wie den Windows-Rechner beispielsweise aufrufen soll, muss es diese natürlich starten können, was mit der *Shell*-Anweisung von VBA (siehe Abschnitt 15.6.6) auch keinerlei Probleme bereitet. Um mehrfache Starts und die damit verbundene Verschwendung von Rechnerressourcen zu vermeiden, sollte das Makro allerdings vorher prüfen, ob die Anwendung nicht bereits läuft. Mit VBA geht das nicht, das Windows-API hält dagegen gleich mehrere Lösungen bereit.

Die meistgenutzte Lösung dürfte der Einsatz der Funktion *FindWindow* sein, die man nach Auskunft der üblichen API-Dokumentationen wie folgt deklarieren muss, um sie in einem Makro nutzen zu können:

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

Wie man sieht, besitzt die *FindWindow*-Funktion zwei Argumente, die der zweifelsfreien Identifikation der gewünschten Anwendung dienen. Das erste Argument *lpClassName* benennt den sogenannten Klassennamen, eine vom jeweiligen Programmierer festgelegte Bezeichnung der zentralen Anwendungskomponente, die beim Windows-Rechner „CalcFrame“ lautet (der Klassenname von Excel ist „XLMAIN“, der von Word „OpusApp“). Kennt man den Klassennamen nicht, gibt sich *FindWindow* auch mit dem Titel des Anwendungsfensters zufrieden, den man im zweiten Argument *lpWindowName* übergibt. Im Fall des Windows-Rechners lautet dieser schlicht „Rechner“.

Findet *FindWindow* ein Anwendungsfenster, das den angegebenen Argumenten entspricht, dann liefert die API-Funktion im Gegenzug dessen „Handle“ zurück. Das ist eine eindeutige Fensternummer, die das aufrufende Makro – ebenfalls laut API-Dokumentation – in einer Variablen vom Typ *Long* entgegenzunehmen hat. Das könnte ungefähr so aussehen:

```
Dim lngWindowHandle As Long
lngWindowHandle = FindWindow(vbNullString, "Rechner")
```

Mit einer simplen *If-Then-Else*-Abfrage wie der folgenden kann das Makro dann schnell die passenden Schlüsse ziehen und angemessen reagieren:

```
If lngWindowHandle = 0 Then
  Shell "calc.exe"
Else
  Dim lngResult As Long
  lngResult = ShowWindow(lngWindowHandle, SW_RESTORE)
  lngResult = SetForegroundWindow(lngWindowHandle)
End If
```

Die *If*-Anweisung prüft den Wert des zurückgelieferten Handles. Ist der gleich null, läuft der Windows-Rechner offensichtlich nicht, da er ja kein (nummeriertes) Anwendungsfenster besitzt. In dem Fall startet das Makro den Rechner per *Shell*-Anweisung. Liegt der Handle-Wert aber über null, gibt es bereits ein Anwendungsfenster, das nur noch aktiviert werden muss. Das erledigt das Makro im *Else*-Abschnitt mithilfe von zwei weiteren API-Funktionen: Die *ShowWindow*-Funktion bringt das Anwendungsfenster zunächst in seine normale Größe (es könnte ja zum Symbol verkleinert sein), so dass es die *SetForegroundWindow*-Funktion in den Vordergrund holen kann (es könnte ja von anderen Fenstern verdeckt sein).

Der Datentyp LongPtr

Wenn Sie das obige Beispiel mit der 32-Bit-Version von Excel ausführen, werden Sie keinerlei Probleme bemerken. Die ergeben sich erst mit der 64-Bit-Version des Kalkulationsprogramms. Auslöser ist *lngWindowHandle*, eine Variable vom Datentyp *Long*, die das von *FindWindow* bereitgestellte Handle des Anwendungsfensters aufnehmen soll. Und das funktioniert in einer 64-Bit-Umgebung nicht, da Handles hier volle 64 Bit beanspruchen, von denen der stets nur 32 Bit „breite“ Datentyp *Long* dann nur noch die Hälfte speichern kann. Das gleiche Problem tritt übrigens auch bei „Pointern“ auf, das sind vom Windows-API gelieferte Variablen, die auf bestimmte Speicheradressen verweisen.

Damit die Zuweisung von Handles und Pointern nun auch ohne kapitalen Absturz in einem 64-Bit-Excel gelingt, hat Microsoft den Sprachumfang von VBA erstmals seit vielen Jahren wieder (und ausschließlich zu diesem Zweck) erweitert. Wichtigste diesbezügliche Errungenschaft der VBA-Version 7.0 ist der „intelligente" Datentyp *LongPtr*. Der passt seine Bit-Breite automatisch an die der verwendeten Excel-Version an: Im Fall eines 32-Bit-Excels speichert er also 32-Bit-Werte, bei einem 64-Bit-Excel nimmt er entsprechend 64-Bit-Werte auf.

Somit eignet sich der Datentyp *LongPtr* ideal für die Aufnahme von Handles und Pointern, die als Argument oder Rückgabewert von API-Funktionen in Erscheinung treten. Wurden die zugehörigen Variablen bislang „As Long" definiert, so müssen diese Definitionen nun also einfach in „As LongPtr" geändert werden, um eine vorhandene Excel-Lösung 64-Bit-tauglich zu machen.

Der *FindWindow*-Aufruf aus unserem Beispiel würde dann so aussehen:

```
' Beispiel 15\DLL64.xlsm
Dim lngWindowHandle As LongPtr
lngWindowHandle = FindWindow(vbNullString, "Rechner")
```

Hinweis

Die vermeintliche „Intelligenz" des Datentyps *LongPtr* ist nichts weiter als ein cleverer Trick. Der besteht darin, sämtliche *LongPtr*-Variablen je nach Bit-Breite der Excel-Version in den real existierenden Datentyp *Long* (32 Bit) beziehungsweise dessen neuesten Kollegen *LongLong* (64 Bit) umzuwandeln.

Das Schlüsselwort PtrSafe

Mit der Makrointernen Umstellung der Variablen *lngWindowHandle* auf den Datentyp *LongPtr* ist es aber nicht getan. Schließlich „weiß" Excel ja noch gar nicht, dass es sich beim Rückgabewert von *FindWindow* um ein Handle mit variabler Bitbreite handelt. Damit der ausführende VBA-Interpreter von diesem Umstand Kenntnis erhält, muss man die Deklarationszeile der API-Funktion ebenfalls wie folgt ändern:

```
' Beispiel 15\DLL64.xlsm
Declare PtrSafe Function FindWindow Lib "user32" Alias _
  "FindWindowA" (ByVal lpClassName As String, ByVal _
  lpWindowName As String) As LongPtr
```

Die Unterschiede zum Original liegen nicht nur am Ende der Deklarationszeile, wo der Datentyp des Rückgabewerts von *Long* in *LongPtr* geändert wurde, sondern auch in der zusätzlichen Verwendung des Schlüsselworts *PtrSafe*. Es informiert den VBA-Interpreter darüber, dass die *Declare*-Anweisung für die 64-Bit-Version von Excel geschrieben wurde. Ohne dieses Schlüsselwort tritt bei Verwendung der *Declare*-Anweisung auf einem 64-Bit-System ein Kompilierungsfehler auf. Bei der 32-Bit-Version von Excel ist das *PtrSafe*- Schlüsselwort optional. Auf diese Weise wird die Funktion vorhandener *Declare*-Anweisungen nicht beeinträchtigt.

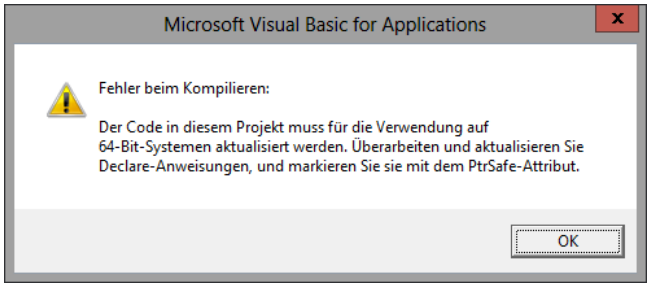

BILD 15.14
In der 64-Bit-Version von Excel müssen *Declare*-Anweisungen zwingend das Schlüsselwort *PtrSafe* enthalten, sonst gibt es Gemecker in Form dieser Fehlermeldung.

Während die Kennzeichnung durch das Schlüsselwort *PtrSafe* also zwingend ist für den 64-Bit-Einsatz, sollte man Datentypänderungen in *Declare*-Anweisungen nur sehr gezielt vornehmen. Die Online-Hilfe von VBA und viele Internetquellen erwecken zwar den Eindruck, als müsste man den Datentyp *sämtlicher* Argumente oder Rückgabewerte von *Long* auf *LongPtr* umstellen. Tatsächlich ist das aber nur bei solchen Argumenten oder Rückgabewerten erforderlich, die für die bilaterale Weitergabe eines Handles oder Pointers verantwortlich sind. Alle anderen Argumente oder Rückgabewerte dürfen ihrem Datentyp *Long* unverändert treu bleiben.

Bei der Deklaration der API-Funktionen *ShowWindow* und *SetForegroundWindow* ist eine Datentypänderung somit nur für das Argument *hwnd* notwendig, das in beiden Fällen das von *FindWindow* gelieferte Handle des Anwendungsfensters übergibt. Als Rückgabewert geben beide Funktionen einen numerischen (Fehler-)Code zurück, der in jedem Fall in einer *Long*-Variablen Platz findet. Die 64-Bit-tauglichen *Declare*-Anweisungen sehen folglich so aus:

```
' Beispiel 15\DLL64.xlsm
Declare PtrSafe Function ShowWindow Lib "user32" (ByVal _
  hwnd As LongPtr, ByVal nCmdShow As Long) As Long
Declare PtrSafe Function SetForegroundWindow Lib "user32" (ByVal _
  hwnd As LongPtr) As Long
```

Da die beiden API-Routinen somit auch in einer 64-Bit-Umgebung „nur“ einen *Long*-Wert zurückgeben, genügt für dessen Aufnahme innerhalb des Makros selbstverständlich auch weiterhin eine Variable desselben Typs. Der entsprechende Aufrufcode in der *If-Then-Else*-Abfrage unseres Beispiels erfordert daher keinerlei Änderungen:

```
' Beispiel 15\DLL64.xlsm
Dim lngResult As Long
lngResult = ShowWindow(lngWindowHandle, SW_RESTORE)
lngResult = SetForegroundWindow(lngWindowHandle)
```

Bedingte Kompilierung

Der Datentyp *LongPtr*, das Schlüsselwort *PtrSafe* und einige andere Elemente, die vollständig in der nachfolgenden Syntaxzusammenfassung dokumentiert sind, sind Neuerungen der VBA-Version 7.0 und stehen damit – vom Datentyp *LongLong* einmal abgesehen – sowohl in der 64- als auch der 32-Bit-Version von Excel zur Verfügung. Somit dürfte jeder VBA-Code, der ursprünglich für die 64-Bit-Fassung von Excel entwickelt wurde und die neuen Elemente

für den Umgang mit API-Funktionen nutzt, auch in der 32-Bit-Version funktionieren. Die Codeverträglichkeit endet allerdings schon bei der Excel-Version 2007, die es ja ausschließlich in einer 32-Bit-Fassung gibt und deren VBA-Version 6.5 Elemente wie *LongPtr*, *PtrSafe* & Co natürlich völlig unbekannt sind.

Man kann trotzdem „allgemeingültigen" VBA-Code schreiben, der API-Funktionen nutzt und dennoch mit allen Bitbreiten und VBA-Varianten bis hinunter zur Version 6.0 (die mit Excel 2000 eingeführt wurde) zurecht kommt. Das Mittel der Wahl heißt „Bedingte Kompilierung" und wurde in Excel 2010/2013 um eine neue Konstante namens *VBA7* erweitert. Die ermöglicht eine saubere Codetrennung für die jüngste VBA-Version und alle VBA-Versionen davor. Das Grundmuster sieht so aus:

```
#If VBA7 Then
  'Anweisungen für VBA 7.0
#Else
  'Anweisungen für frühere VBA-Versionen
#End If
```

Wird der obige Code in Excel (egal, ob 32 oder 64 Bit) ausgeführt, pickt sich der VBA-Interpreter exklusiv die Anweisungen heraus, die im *#If*-Abschnitt durch die Kompilierungskonstante *VBA7* gekennzeichnet sind. Älteren VBA-Interpretern ist die Konstante unbekannt; sie verzweigen daher automatisch in den *#Else*-Abschnitt des Codeblocks.

Im Fall unseres Beispiels würde man die bedingte Kompilierung unter anderem für eine versionsgerechte Deklaration der beteiligten API-Funktionen einsetzen, und zwar so:

```
' Beispiel 15\DLL64.xlsm
#If VBA7 Then
  Declare PtrSafe Function FindWindow Lib "user32" Alias _
    "FindWindowA" (ByVal lpClassName As String, ByVal _
    lpWindowName As String) As LongPtr
  Declare PtrSafe Function ShowWindow Lib "user32" _
    (ByVal hwnd As LongPtr, ByVal nCmdShow As Long) As Long
  Declare PtrSafe Function SetForegroundWindow Lib _
    "user32" (ByVal hwnd As LongPtr) As Long
#Else
  Declare Function FindWindow Lib "user32" Alias _
    "FindWindowA" (ByVal lpClassName As String, ByVal _
    lpWindowName As String) As Long
  Declare Function ShowWindow Lib "user32" (ByVal hwnd As _
    Long, ByVal nCmdShow As Long) As Long
  Declare Function SetForegroundWindow Lib "user32" (ByVal _
    hwnd As Long) As Long
#End If
```

Darüber hinaus käme die bedingte Kompilierung auch innerhalb des Makros für die Aktivierung des Windows-Rechners zum Einsatz, und zwar bei der Deklaration der Variablen *lngWindowHandle*:

```
' Beispiel 15\DLL64.xlsm
Public Sub RechnerAktivierenOderStarten()
    #If VBA7 Then
        Dim lngWindowHandle As LongPtr
    #Else
        Dim lngWindowHandle As Long
    #End If

    lngWindowHandle = FindWindow(vbNullString, "Rechner")
    If lngWindowHandle = 0 Then
        Shell "calc.exe"
    Else
        Dim lngResult As Long

        lngResult = ShowWindow(lngWindowHandle, SW_RESTORE)
        lngResult = SetForegroundWindow(lngWindowHandle)
    End If
End Sub
```

Hinweis

Das vollständige Beispiel finden Sie in der Datei *DLL64.xlsm* im Unterordner 15 der Beispieldateien.

15.7.3 Syntaxzusammenfassung

Neue Elemente von VBA 7.0 für die 64-Bit-Programmierung	
CLngLng	konvertiert einen Wert in den Datentyp LongLong
CLngPtr	konvertiert einen Wert in den Datentyp LongPtr
LongLong	8-Byte-Datentyp, der nur in 64-Bit-Versionen von Excel 2010/2013 zur Verfügung steht
LongPtr	variabler Datentyp, der auf 32-Bit-Versionen von Excel 2010/2013 als 4-Byte-Datentyp (Long) und auf 64-Bit-Versionen als 8-Byte-Datentyp (LongLong) ausgelegt ist
ObjPtr	Objektkonverter; gibt auf 64-Bit-Versionen LongPtr und auf 32-Bit-Versionen Long zurück
PtrSafe	gibt an, dass die Declare-Anweisung mit 64-Bit-Systemen kompatibel ist
StrPtr	Zeichenfolgenkonverter; gibt auf 64-Bit-Versionen LongPtr und auf 32-Bit-Versionen Long zurück
VarPtr	Variantenkonverter; gibt auf 64-Bit-Versionen LongPtr und auf 32-Bit-Versionen Long zurück
VBA7	Konstante, die die bedingte Kompilierung von Anweisungblöcken für VBA 7 und ältere VBA-Versionen ermöglicht

15.9 Apps für Office

Hinter dem neuen Befehl Apps für Office im Menüband Einfügen von Excel 2013 verbirgt sich ein radikal neues Erweiterungskonzept, das Apps an die Stelle von Makros setzen und Webtechniken und Cloud-Dienste direkt in die Bedienoberflächen aller Office-Anwendungen (und ihrer jeweiligen Web-App-Versionen) integrieren will.

Obwohl sich Apps für Office (oder kürzer: Office-Apps) auch lokal bereitstellen lassen – was wir in diesem Kapitel anhand von zwei Beispielen demonstrieren werden –, erhält man sie vorzugsweise im *Office Store*. Das ist ein gleichfalls neuer Online-Shop, den man direkt aus den Office-Anwendungen heraus erreicht.

Der Office Store präsentiert das bislang vorwiegend englischsprachige App-Angebot in übersichtlicher Form und reduziert die Installation des gewählten Helferleins auf wenige Mausklicks. Die schnelle Verfügbarkeit macht Office-Apps für Anwender und damit natürlich auch für Entwickler attraktiv. Grund genug also, sich mit den Grundlagen der neuen Office-Apps vertraut zu machen.

15.9.1 Bestandteile einer Office-App

Eine Office-App ist im Grunde eine normale Webanwendung, die aber nicht auf einem Server im Internet, sondern in Office „gehostet“ und angezeigt wird. Die typische Office-App besteht aus den folgenden Komponenten:

- **Webseite:** Dabei handelt es sich um eine (halbwegs) normale HTML-Datei, die man wie jede andere Webseite auch mit Steuerelementen aus dem HTML-, ASP.NET-, Silverlight- oder Flash-Fundus bestücken kann. Das Aussehen der Webseite kann der Entwickler über standardmäßige Webtechniken wie HTML5, XML und CSS3 bestimmen. Die Webseite bildet die Bedienoberfläche der Office-App.
- **Skriptdatei:** Das ist zumeist eine JavaScript-Datei, die den Programmcode der App enthält und damit für die „Action“ zuständig ist. Wie bei regulären Webanwendungen wird die Verbindung zwischen Webseite und Skriptdatei häufig über das *onclick*-Attribut der Steuerelemente hergestellt. Es weist dem jeweiligen Steuerelement eine Ereignisroutine in der Skriptdatei zu. Der Code darin bestimmt, was beim Anklicken des Steuerelements geschieht.

  Grundsätzlich lässt sich die gewünschte Funktionalität einer Office-App aber nicht nur über eine externe Skriptdatei, sondern über *jede* Art der client- oder serverseitigen Programmierung bereitstellen. Das kann im einfachsten Fall ein in die Webseite eingebettetes Skript sein, in anspruchsvolleren Szenarien eine komplexe ASP.NET-Anwendung. Über REST-APIs kann eine Office-App so gut wie jeden Web-Service (siehe Abschnitte 15.4 und 15.8.4) zur Informationsbeschaffung anzapfen.
- **Icon-Datei** (optional): Diese BMP-, GIF-, EXIF-, JPG-, PNG- oder TIFF-Datei enthält das Icon der Office-App, das eine Größe von 32 mal 32 Pixel aufweisen muss. Fehlt die Icon-Datei oder weist sie eine andere Bildgröße auf, erhält die App ein monochromes Standard-Icon zugewiesen.

- **OfficeStyles.css** (optional): Diese Datei stellt das Stylesheet für die Office-App bereit und weist allen Bestandteilen der Webseite die Office-typischen Schriftarten und Formatierungen zu.
- **Manifestdatei:** Diese XML-Datei ist das Bindeglied zwischen den einzelnen App-Elementen. Die Manifestdatei verrät der Office-Anwendung unter anderem, wo die Webseiten- und die Icon-Datei (falls vorhanden) zu finden sind. Darüber hinaus definiert sie die Einstellungen der App, ihre Fähigkeiten und Rechte.
- **JavaScript-API für Office:** Diese von Microsoft bereitgestellte und online verfügbare Bibliothek (die eine JavaScript-Datei ist) stellt die Verbindung zwischen der Office-Anwendung und der App-Webseite her. Die Bibliothek sorgt dafür, dass die App unter anderem auf Inhalte des Dokuments zugreifen oder mit der als Host fungierenden Office-Anwendung kommunizieren kann.

  Die Risiken einer solchen Interaktion zwischen Anwendung und App will Microsoft insbesondere durch eine sorgsame Trennung der beiden klein halten. Dazu sperrt der Hersteller die HTML-Datei in ein Webbrowser-Control und lässt dieses in einem von der Host-Anwendung unabhängigen Prozess ausführen. Zu den weiteren Sicherheitsmaßnahmen gehört eine restriktive Laufzeitumgebung. Die überwacht jede Interaktion über Prozessgrenzen hinweg und gestattet Zugriffe auf die Daten und die Bedienoberfläche der Office-Anwendung grundsätzlich nur über asynchrone Methoden, die den Office-Prozess weder ausbremsen noch zum Entgleisen bringen können.

Die Auflistung der App-Elemente lässt bereits erahnen, dass die Entwicklung von Office-Apps vorzugsweise in die Zuständigkeit erfahrener Webentwickler fällt. Klassische Office-Entwickler, die sich „nur“ mit VBA und eventuell noch mit den Visual Studio Tools for Office (VSTO, siehe Abschnitt 15.8) auskennen, werden sich in einer neuen Programmierwelt zurechtfinden müssen.

Aus diesem Grund kann das vorliegende Kapitel auch kaum mehr als eine Einführung in das Thema sein. Für das Verständnis der folgenden Ausführungen und Codepassagen sind grundlegende HTML-, XML- und JavaScript-Kenntnisse erforderlich. Wer weiterführende Infos benötigt, findet diese auf Microsofts Portal für Office-App-Entwickler *[Link 44]*:

http://msdn.microsoft.com/en-us/Office/apps

15.9.2 Typen von Office-Apps

Es gibt drei Typen von Office-Apps, die man nach dem Erscheinungsort der App-Oberfläche (die ja von der App-Webseite bereitgestellt wird) unterscheidet:

- **Aufgabenbereich-App:** Diese Office-App zeigt sich im Aufgabenbereich der Office-Anwendung am rechten Rand des Dokumentfensters. Apps dieser Art eignen sich insbesondere dazu, dem Anwender kontextbezogene Informationen und Funktionen – für die Suche oder das Übersetzen von Inhalten beispielsweise – zu liefern.
- **Inhalt-App:** Diese Office-App erscheint ähnlich wie ein Excel-Diagramm als abgegrenzter Bereich innerhalb des Dokuments. Inhalt-Apps eignen sich vor allem für die Visualisierung von Daten oder die Wiedergabe von YouTube-Videos und anderen Internetmedien.

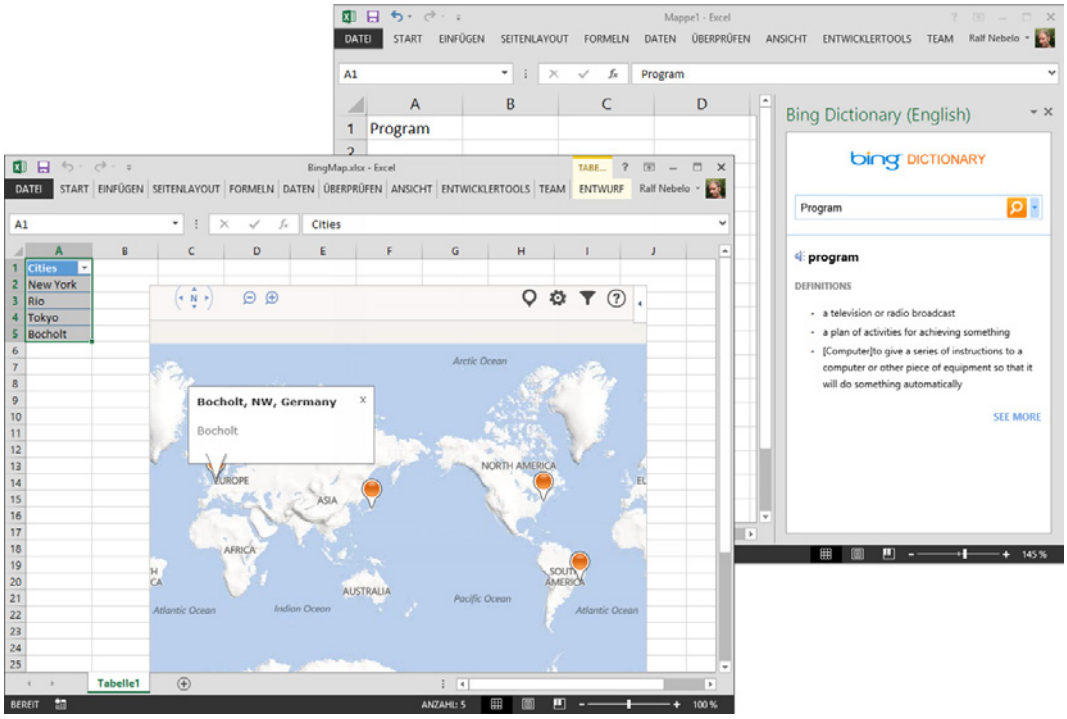

BILD 15.26 Microsofts Suchmaschine Bing stellt Excel-Anwendern unter anderem eine Aufgabenbereich-App (hinten) sowie eine Inhalt-App zur Verfügung.

- **Mail-App:** Sie klinkt sich in Outlook-Formulare ein und stellt dem Anwender dort maßgeschneiderte Infos und Funktionen bereit, die sich auf das jeweils angezeigte Outlook-Element – eine Nachricht, eine Besprechungsanfrage oder einen Termin etwa – beziehen. Mail-Apps funktionieren nur im Zusammenspiel mit Exchange 2013, normale POP- und IMAP-Konten werden nicht unterstützt.

Während Mail-Apps quasi naturgemäß auf Outlook festgelegt sind, sollten Inhalt- und Aufgabenbereich-Apps von der Konzeption her eigentlich in allen Office-Anwendungen nutzbar sein, die der Bearbeitung von Dokumenten dienen. Momentan unterstützt aber nur Excel beide App-Typen, während sich PowerPoint, Project und Word 2013 nur mit Aufgabenbereich-Apps erweitern lassen. Die Tabelle zeigt den aktuellen Stand, der sich jedoch jederzeit ändern kann.

Anwendung	Aufgabenbereich-App	Inhalt-App	Mail-App
Access 2013	-	-	-
Excel 2013	+	+	-
Excel-Web-App	-	+	-
Outlook 2013	-	-	+
Outlook-Web-App	-	-	+
PowerPoint 2013	+	-	-

Anwendung	Aufgabenbereich-App	Inhalt-App	Mail-App
PowerPoint-Web-App	-	-	-
Project 2013	+	-	-
Word 2013	+	-	-
Word-Web-App	-	-	-

Anders als ein konventionelles Office-Add-in, das man für jede Applikation separat entwickeln muss, lässt sich eine App aber sehr leicht so gestalten, dass sie mit demselben Code in jeder Office-Anwendung funktioniert, die den jeweiligen App-Typ unterstützt.

15.9.3 Werkzeuge für die App-Entwicklung

Auch wenn die App-Entwicklung einiges Wissen über Webtechniken erfordert, so benötigt man doch erfreulich wenig Handwerkszeug dafür: Ein schlichter Texteditor genügt für den Anfang (und die Realisierung der nachfolgenden Beispiel-Apps).

Allerdings sollte man es auch nicht übertreiben, indem man sich mit dem Windows-eigenen Notepad begnügt. Wesentlich angenehmer lässt es sich beispielsweise mit *Notepad++* arbeiten, da es unter anderem mehrere Fenster für die Bearbeitung der verschiedenen Projektdateien unterstützt. Die Download-Adresse für das nützliche Open-Source-Werkzeug lautete zuletzt wie folgt *[Link 40]*:

http://notepad-plus-plus.org

Wer es komfortabler mag, verwendet die *Napa Office 365 Development Tools*. Dabei handelt es sich um eine webbasierte Entwicklungsumgebung, mit der Sie im Browser Projekte erstellen, Code schreiben und Ihre Apps ausführen können. Weitere Informationen finden Sie hier *[Link 45]*:

http://msdn.microsoft.com/de-de/library/Office/apps/jj220038

Das mit Abstand komfortabelste, leistungsfähigste, aber auch teuerste Werkzeug zum Erstellen einer Office-App ist *Visual Studio 2012*. Die Entwicklungsumgebung verfügt über eine spezielle *App für Office*-Projektvorlage, die dem Entwickler viele Handgriffe, die er ansonsten manuell erledigen müsste, erspart. Darüber hinaus beschleunigt Visual Studio die App-Entwicklung mit einer Projektmappe, die eine vorkonfigurierte Manifestdatei, Skriptbibliotheken, Stylesheets sowie HTML- und JavaScript-Ausgangsdateien enthält.

15.9.4 Beispiel 1: SimpleApp

Die Arbeit an unserer ersten Beispiel-App beginnt mit dem Anlegen einer Ordnerstruktur, die die diversen Projektdateien aufnimmt. Dazu benötigen Sie eine Netzwerkfreigabe, die auf einem Rechner Ihres Firmen- respektive Heimnetzes oder einer Netzwerkfestplatte (NAS) liegen kann. Lokale Festplatten kommen nicht in Betracht, da Office 2013 für Speicherortangaben ausschließlich echte URLs (ohne „file:///“-Präfix) akzeptiert.

Fügen Sie der Netzwerkfreigabe zunächst ein neues Stammverzeichnis für die App-Entwicklung hinzu, das Sie mit *OfficeApps* benennen. Dieses Stammverzeichnis dient unter anderem der Aufnahme der Manifestdatei(en) (und ist damit gleichzeitig auch der Manifestordner). Fügen Sie dem Stammverzeichnis einen Unterordner namens *SimpleApp* hinzu, der die Heimat der Beispiel-App bildet.

Die Webseite von SimpleApp

Zunächst erstellen Sie die Webseite der Beispiel-App, indem Sie eine HTML-Datei mit folgendem Inhalt anlegen (die erste Zeile bitte weglassen, sie verweist nur auf den Speicherort der Datei auf der Buch-CD). Speichern Sie die HTML-Datei anschließend unter dem Namen *SimpleApp.html* im Unterordner *SimpleApp*.

```
<! 15\OfficeApps\SimpleApp\SimpleApp.html >
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=Edge"/>
    <link rel="stylesheet" type="text/css"
      href="../OfficeStyles.css" />
    <script src="SimpleApp.js"></script>
  </head>
  <body>
    <input type="text" value=http://www.hanser-fachbuch.de
      id="TextBox1" style="margin-top: 10px; width: 280px" />
    <input type="button" value="Öffnen" id="Button1" onclick=
      "doAction()" style="margin-top: 10px; margin-right: 10px;
      padding: 0px; width: 100px;" />
  </body>
</html>
```

Wer sich mit HTML ein wenig auskennt, erkennt rasch, dass sich die Struktur dieser App-Webseite kaum von jeder anderen Webseite unterscheidet. Erklärungsbedürftig ist allenfalls das zweite *meta*-Element innerhalb des HTML-Kopfs, das den Internet Explorer zur bevorzugten Rendering-Engine erhebt.

Das *link*-Element stellt einen Verweis auf das Stylesheet *OfficeStyles.css* im übergeordneten Verzeichnis (..\) her. Da diese CSS-Datei ziemlich umfangreich ist, kopieren Sie sie der Einfachheit halber von der Buch-CD in Ihren neu angelegten *OfficeApps*-Ordner, wo sie künftig allen Apps zur Verfügung steht.

Das *script*-Element erhebt die Datei *SimpleApp.js* im gleichen Verzeichnis in den Rang der zuständigen Skriptdatei.

Die Anweisungen innerhalb des *body*-Blocks statten die Webseite mit zwei HTML-Controls aus, einer Textbox und einem Button. In der Textbox erscheint die Webadresse „*http://www.hanser-fachbuch.de*“ als Vorgabewert, dem Button wird per *onclick*-Attribut eine Funktion namens „doAction“ als Ereignisroutine für das Anklicken zugewiesen.

Die Skriptdatei von SimpleApp

Die oben erwähnte *doAction*-Funktion bildet den einzigen Inhalt der nachfolgend abgedruckten Skriptdatei *SimpleApp.js*. Speichern Sie diese – genau wie die HTML-Datei zuvor – im Unterordner *SimpleApp*. Die erste Zeile können Sie beim Abtippen wieder weglassen.

```
/* 15\OfficeApps\SimpleApp\SimpleApp.js */
function doAction() {
  var url = document.getElementById("TextBox1").value
  window.location.href = url;
}
```

Der Inhalt der *doAction*-Funktion ist schnell erklärt. Die erste Anweisungszeile liest den aktuellen Inhalt der Textbox, bei dem es sich um eine gültige Webadresse handeln sollte, in die Variable *url* ein. Die zweite Zeile öffnet dann ein Browser-Fenster, das den Inhalt der entsprechenden Internetseite anzeigt.

Die Manifestdatei von SimpleApp

Die Manifestdatei unserer Beispiel-App hat folgenden Inhalt (erste Zeile bitte wieder weglassen):

```
<! 15\OfficeApps\SimpleApp.xml >
<?xml version="1.0" encoding="utf-8"?>
<OfficeApp xmlns=
  "http://schemas.microsoft.com/Office/appforOffice/1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:type="TaskPaneApp">
  <Id>08afd7fe-1631-42f4-84f1-5ba51e242f11</Id>
  <Version>1.0</Version>
  <ProviderName>Ralf Nebelo</ProviderName>
  <DefaultLocale>EN-US</DefaultLocale>
  <DisplayName DefaultValue="SimpleApp"/>
  <Description DefaultValue="Meine erste Office-App"/>
  <IconUrl DefaultValue=
    "\\MYBOOKDATA\Public\OfficeApps\SimpleApp\SimpleApp.png"/>

  <Capabilities>
    <Capability Name="Document"/>
    <Capability Name="Workbook"/>
  </Capabilities>

  <DefaultSettings>
    <SourceLocation DefaultValue=
      "\\MYBOOKDATA\Public\OfficeApps\SimpleApp\SimpleApp.html"/>
  </DefaultSettings>
  <Permissions>ReadWriteDocument</Permissions>
</OfficeApp>
```