Excel 2016 programmieren

Abläufe automatisieren, (Office-)Add-ins und Anwendungen entwickeln

Bearbeitet von
Ralf Nebelo, Michael Kofler

1. Auflage 2016. Buch. 880 S.
ISBN 978 3 446 44798 1
Format (B x L): 18,5 x 24,5 cm
Gewicht: 1696 g

HANSER

Leseprobe

Michael Kofler, Ralf Nebelo

Excel 2016 programmieren

Abläufe automatisieren, (Office-)Add-ins und Anwendungen entwickeln

ISBN (Buch): 978-3-446-44798-1

ISBN (E-Book): 978-3-446-45008-0

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-44798-1

sowie im Buchhandel.

Inhalt

Vorwort

Excel bietet von Haus aus ein riesiges Spektrum von Funktionen. Wozu sollten Sie dann noch selber Makros, Add-ins diverser Art und andere Programmerweiterungen mit VBA, Visual Studio oder anderen Werkzeugen entwickeln? Weil Sie damit …

- … eigene Tabellenfunktionen programmieren können, die einfacher anzuwenden sind als komplizierte Formeln.
- … Excel nach Ihren Vorstellungen konfigurieren und auf diese Weise eine einfachere und effizientere Programmbedienung erreichen können.
- … komplexe Arbeitsschritte wie etwa das Ausfüllen von Formularen durch „intelligente" Formulare (alias Mustervorlagen) strukturieren und erleichtern können.
- … immer wieder auftretende Arbeitsvorgänge automatisieren können. Das empfiehlt sich besonders dann, wenn regelmäßig große Datenmengen anfallen, die verarbeitet, analysiert und grafisch aufbereitet werden sollen.
- … eigenständige und leistungsfähige Excel-Lösungen erstellen können, die sich durch maßgeschneiderte Bedienelemente nahtlos in das Menüband, die sogenannte Backstage-Ansicht (im Datei-Menü) oder andere Teile der Excel-Oberfläche integrieren.

  Damit lassen sich Excel-Anwendungen in ihrer Bedienung so weit vereinfachen, dass sie von anderen Personen (auch von Excel-Laien) ohne lange Einweisung verwendet werden können.

Das notwendige Know-how für alle diese Formen der Excel-Programmierung finden Sie in diesem Buch. *Übrigens: Auch wenn Excel 2016 auf dem Titel steht, so gilt das Gesagte – oder besser: Geschriebene – doch für alle Programmversionen ab 2007 (und zum größten Teil auch für die Versionen davor).*

Wenn es Dinge gibt, die in einer älteren Version anders funktionieren als in Excel 2016, so wird das ausdrücklich erwähnt. Falls das wider Erwarten einmal nicht der Fall sein sollte, bitten wir schon jetzt um Verzeihung. Bei so vielen Versionen verlieren auch erfahrene Autoren manchmal den Überblick.

2007 bis 2016 – Neues in Excel

Mit der radikal neuen Multifunktionsleiste, die die früheren Menüs und Symbolleisten plötzlich sehr alt aussehen ließ (und letztlich in Rente schickte), war Excel 2007 eine echte Revolution. Excel 2010 ließ es entwicklungstechnisch deutlich ruhiger angehen und bescherte uns statt einer großen *Re*volution viele kleine *E*volutionen.

Eine davon war der neue *Oberflächeneditor*, mit dem wir nicht mehr nur die unscheinbare „Symbolleiste für den Schnellzugriff" nach unseren Wünschen konfigurieren dürfen, sondern die komplette Multifunktionsleiste. Die heißt nun übrigens *„Menüband"* (siehe Abschnitt 8.2) und beschränkt sich auf solche Befehle, die der Bearbeitung von Dokumentinhalten dienen. Für alle anderen Befehle, die das Dokument als Ganzes betreffen (Speichern, Drucken etc.), hat Microsoft die sogenannte *Backstage-Ansicht* (siehe Abschnitt 8.5) erfunden, die das Office-Menü von Excel 2007 ersetzt. Menüband und Backstage-Ansicht bilden seither die Kommandozentrale von Excel und zeichnen sich durch eine konsequente Aufgabenteilung aus.

Konsequenz zeigte Microsoft auch bei der *Oberflächenprogrammierung*. Hier gilt seit Excel 2010 für alle Bestandteile – Menüband, Backstage-Ansicht, Symbolleiste für den Schnellzugriff *und* Kontextmenüs – das gleiche „duale Prinzip": XML-Code bestimmt das Design, VBA-Code die Funktion. Mit dem Know-how, das Sie sich womöglich schon bei der Anpassung der früheren Multifunktionsleiste erworben haben, können Sie jetzt also die gesamte Excel-Oberfläche verändern und eigene Lösungen integrieren (Kapitel 8).

Evolutionär präsentierte sich Excel 2010 auch bei der Visualisierung von Zahlen. So fanden die *SmartArt-Diagramme* (Abschnitt 10.7), die mit der Version 2007 eingeführt wurden, Eingang in das Objektmodell, so dass man sie nun programmatisch erstellen oder verändern kann. Darüber hinaus hat Excel 2010 der Welt die sogenannten *Sparklines-Diagramme* (Abschnitt 10.6) beschert, ein seinerzeit völlig neuer und ebenfalls programmierbarer Diagrammtyp, der in eine einzelne Zelle passt und sich insbesondere für die Visualisierung von Trends eignet.

Wo Licht ist, ist bekanntlich auch Schatten. Und das gilt insbesondere für die Tatsache, dass es Excel seit der Version 2010 auch in einer *64-Bit-Version* zu kaufen gibt. Dass die nicht nur Vorteile hat, sondern auch massive Nachteile in Form von diversen Inkompatibilitäten, zeigt der Abschnitt 15.7 (und was Sie dagegen tun können, natürlich auch).

Excel 2013 präsentierte sich dem Anwender erstmals in einem nüchternen, von Schatten und Transparenzeffekten befreiten Look, der sich an der Optik von Windows 8 orientierte. Und dazu passend fand sich eine neuerlich aufgeräumte und entschlackte Menüband- und Backstage-Oberfläche, in der man so manchen Befehl aus früheren Versionen leider nicht mehr finden konnte.

Als Ausgleich gab es neue Funktionen wie *Schnellanalyse* und *Empfohlene Diagramme*, die die Erstellung von Diagrammen beschleunigten. Arbeitsmappen ließen sich standardmäßig „in der Cloud" und somit online speichern, manche Diagramme in animierter Form anzeigen und Pivot-Tabellen auf der Basis mehrerer Listen beziehungsweise Tabellen generieren. Unter der Haube gab es die eine oder andere neue Tabellenfunktion zu entdecken, unter anderem für das direkte Anzapfen von Webdiensten (siehe Abschnitt 15.4).

Der wichtigste und aus Entwicklersicht interessanteste Neuzugang aber war die *App für Office*. Dabei handelte es sich um ein seinerzeit völlig neues Erweiterungskonzept, das Webtechniken an die Stelle von VBA-Makros setzen wollte (und weiterhin will). Wie (und ob) das funktioniert, ist detailliert in Kapitel 15.9 beschrieben.

Und was gibt es Neues in der jüngsten Excel-Version 2016? Aus Anwendersicht wären da wohl vor allem stark verbesserte Funktionen für die gemeinsame (Echtzeit-)Arbeit an Dokumenten zu nennen, die Anpassung an Windows 10 und Touch-Bedienung sowie die neue

Hilfefunktion *„Was möchten Sie tun?“*, die den User ohne Umweg über wortreiche Schritt-für-Schritt-Anleitungen direkt zur gesuchten Funktion führt.

Aus der Sicht des Programmierers sind andere Neuerungen aber viel interessanter. So gibt es jetzt eine Reihe neuer Diagrammtypen wie *Wasserfall-*, *Pareto-* oder *Treemap*-Diagramme, die nicht nur visuell überzeugen, sondern sich auch noch programmatisch generieren und verändern lassen. Wie das funktioniert, haben wir in Kapitel 10.8 detailliert beschrieben.

Ein weiteres Highlight: Microsoft hat die Funktionalität des ehemaligen *Power-Query*-Add-ins nun vollständig in Excel (und dessen Objektmodell) integriert. Damit steht uns Entwicklern jetzt ebenfalls ein extrem mächtiges Tool zur Verfügung, mit dem sich Daten aus nahezu jeder Datenquelle auswählen, aufbereiten und in ein Excel-Arbeitsblatt importieren lassen. Kapitel 12.2.1 vermittelt Ihnen das dazu notwendige Know-how.

Natürlich hat es auch wieder die eine oder andere Änderung im Vergleich zu früheren Excel-Versionen gegeben – was immer etwas Verwirrung stiftet. So heißen die frisch vorgestellten *Apps für Office* (alias *Office-Apps*) nun plötzlich *Office-Add-ins*, was dazu führt, dass wir es in Excel 2016 erstmals mit drei verschiedenen Arten von Add-ins zu tun haben, nämlich „Excel-Add-ins“ (siehe Kapitel 15.1), „COM-Add-ins“ (Kapitel 15.6 und 15.8) sowie besagte „Office-Add-ins“. Bei Letzteren hat es neben der neuen Namensgebung auch diverse Änderungen und Erweiterungen in Sachen Programmierung gegeben. Die Details finden Sie in Kapitel 15.9.

Warum dieses Buch?

Im Gegensatz zu anderen Titeln, die sich mit dem Thema Excel-Programmierung beschäftigen, liefert Ihnen dieses Buch *keine* systematische Beschreibung von Objekten, ihren Eigenschaften und Methoden oder VBA-Befehlen. Wer so etwas sucht, ist mit der Hilfefunktion des VBA-Editors und mit zahlreichen Internetquellen besser bedient.

Anstelle einer umfassenden Referenz stehen bei diesem Buch praktische Lösungen und Anwendungsmöglichkeiten im Vordergrund. Die zugehörigen Code-Beispiele lassen sich relativ leicht an eigene Bedürfnisse anpassen, was die Entwicklungszeit für manches berufliche oder private Programmiervorhaben spürbar verkürzen kann.

Dass man bei praxisbezogenen Projekten natürlich auch sehr viel über Objekte (die wichtigsten sogar!), vor allem aber über sinnvolle Formen ihres programmierten Zusammenarbeitens erfährt, ist quasi ein Nebeneffekt. Gleichzeitig nennen wir aber auch die Probleme Excels beim Namen, um Ihnen die langwierige Suche nach Fehlern zu ersparen, die Sie gar nicht selbst verursacht haben.

Neben konkreten Programmierlösungen liefert Ihnen dieses Buch aber auch sehr viel Insider-Wissen über die Bedienung von Excel. Damit werden Sie so ganz nebenbei zum „Power-User“ und können so manches Anwendungsproblem mit ein paar Mausklicks lösen, für das Sie ansonsten womöglich ein Programm geschrieben hätten … ;-)

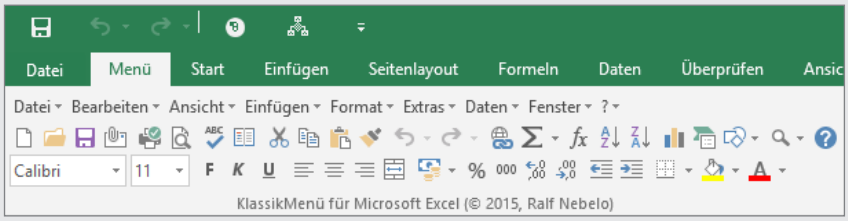
Jenseits von VBA

Obwohl VBA immer noch das wichtigste Werkzeug für die Entwicklung von Excel-Lösungen ist (und daher im Mittelpunkt dieses Buchs steht), stellen sich zunehmend mehr Aufgaben, die mit der „eingebauten“ Programmiersprache des Kalkulationsprogramms nur noch teilweise oder gar nicht mehr zu lösen sind. Beispiele sind etwa die Anpassung von Menüband

(siehe Abschnitt 8.2.2) und Backstage-Ansicht (8.5.1), die Programmierung individueller Aufgabenbereiche (15.8.4.1), die Abfrage von Web Services (15.4) oder die Integration von Webtechniken in Form der neuen Office-Add-ins (15.9).

Damit Sie solche Aufgaben dennoch meistern können, stellt Ihnen dieses Buch die erforderlichen Werkzeuge vor und liefert Ihnen das notwendige Know-how für den erfolgreichen Einsatz. Das erspart Ihnen mühsame Recherchen im Internet, den Kauf weiterer Bücher und lässt Sie mitunter auch kleine programmiertechnische „Wunder" vollbringen – die Wiederbelebung der mit Excel 2003 „entschlafenen" Menüs und Symbolleisten (siehe Abschnitt 8.2.5) beispielsweise. Darüber dürften sich insbesondere altgediente Excel-Anwender freuen, die sich selbst in der nunmehr vierten Ribbon-Version von Excel noch immer nicht im Menüband zurechtfinden.

Tipp



Die Datei *KlassikMenü.xlam* im Unterordner 8 der Beispieldateien enthält eine vollständige Nachbildung der Menü- und Symbolleiste von Excel 2003. Sie können diese Datei als sofort nutzbares Add-in in Excel ab Version 2007 einbinden. Abschnitt 8.2.6 verrät, wie Sie dazu vorgehen müssen.

Viel Erfolg!

Die Beispiele dieses Buchs zeigen, wie weit Excel-Programmierung gehen kann. Die Möglichkeiten sind beinahe unbegrenzt! Wer sie nutzen will, muss sich aber nicht mehr nur im komplexen Objektmodell von Excel und in VBA zurechtfinden, sondern zunehmend auch in angrenzenden Programmierwelten.

Dabei will Ihnen dieses Buch eine praktische Orientierungshilfe sein. Mit zunehmender Übersicht und Erfahrung beginnt dann die Office-Programmierung mit VBA, Visual Studio, XML, JavaScript und diversen anderen Werkzeugen richtig Spaß zu machen.

Und wenn das der Fall ist, lässt auch der gewünschte Erfolg nicht lange auf sich warten. Genau den wünschen wir Ihnen von Herzen!

Michael Kofler und *Ralf Nebelo*, März 2016

http://www.kofler.info

Konzeption des Buchs

Visual Basic für Applikationen (oder kurz: VBA) ist eine sehr leistungsfähige Programmiersprache. Die große Zahl von Schlüsselwörtern bringt aber auch viele Probleme mit sich. Während des Einstiegs ist es so gut wie unmöglich, auch nur halbwegs einen Überblick über VBA zu gewinnen. Und selbst nach monatelanger Programmierung mit VBA wird die Hilfe der wichtigste Ratgeber zu den Details eines bestimmten Schlüsselworts bleiben. Dieses Buch versucht deswegen ganz bewusst, das zu bieten, was in der Originaldokumentation bzw. in der Hilfe zu kurz kommt:

- detaillierte Informationen für die Entwicklung eigener VBA-Lösungen und deren Integration in Menüband, Backstage-Ansicht und andere Bestandteile der Excel-Oberfläche,
- „echte" Anwendungen in Form von konkreten, realitätsbezogenen Beispielen,
- themenorientierte Syntaxzusammenfassungen (z. B. alle Eigenschaften und Methoden zur Bearbeitung von Zellbereichen),
- aussagekräftige Beschreibungen der wichtigsten Objekte von VBA und ihre Einordnung in die Objekthierarchie.

Darüber hinaus liefert Ihnen dieses Buch sehr viel Know-how für fortgeschrittene Programmierthemen, bei denen VBA nicht unbedingt im Mittelpunkt steht:

- Einsatz von DLL-Funktionen,
- ActiveX-Automation,
- Programmierung eigener Add-ins,
- Verwendung von Web Services,
- 64-Bit-Programmierung,
- Realisierung von Office-Anwendungen mit den Visual Studio Tools for Office (VSTO),
- Entwicklung von Office-Add-ins mit Hilfe von Webtechnologien.

Einem Anspruch wird das Buch aber ganz bewusst nicht gerecht: dem der Vollständigkeit. Es erscheint uns sinnlos, Hunderte von Seiten mit einer Referenz aller Schlüsselwörter zu füllen, wenn Sie beinahe dieselben Informationen auch in der Hilfe finden können. Anstatt zu versuchen, auch nur den Anschein der Vollständigkeit zu vermitteln, haben wir uns bemüht, wichtigeren Themen den Vorrang zu geben und diese ausführlich, fundiert und praxisorientiert zu behandeln.

Formalitäten

Die Namen von Menüs, Befehlsregisterkarten, Symbolen, Buttons und anderer Dialog- und Oberflächenelemente werden in Kapitälchen dargestellt: Datei | Öffnen, Abbruch oder OK. Die Anweisung Überprüfen | Blatt schützen | Zellen formatieren meint, dass Sie zuerst die Befehlsregisterkarte Überprüfen öffnen, den Befehl Blatt schützen anklicken und im daraufhin erscheinenden Dialog das Kontrollkästchen Zellen formatieren auswählen sollen.

VBA-Schlüsselwörter, Variablen- und Prozedurnamen sowie Datei- und Verzeichnisnamen werden *kursiv* angegeben, etwa *Application*-Objekt, *Visible*-Eigenschaft, *strName*-Variable oder *C:\Muster.xlsm*. Tabellenfunktionen wie *WENN()* erscheinen in der gleichen Schrift,

aber in Großbuchstaben. (Tabellenfunktionen sind auch anhand der Sprache von VBA-Schlüsselwörtern zu unterscheiden: VBA-Schlüsselwörter sind grundsätzlich englisch, Tabellenfunktionsnamen immer deutsch.)

Beispielcode, Beispieldateien, Download-Dateien zum Buch

Aus Platzgründen sind in diesem Buch immer nur die wichtigsten Code-Passagen der Beispielprogramme abgedruckt. Den vollständigen Code finden Sie unter Download-Dateien zum Buch, die Sie unter der folgenden Internetadresse herunterladen können:

http://downloads.hanser.de/

Die Beispieldateien sind in Verzeichnissen angeordnet, deren Namen den Kapitelnummern entsprechen. VBA-Code in diesem Buch beginnt immer mit einem Kommentar im Format Verzeichnis\Dateiname, der auf die entsprechende Beispieldatei verweist:

```
' 01\format.xlsm
Sub FormatAsResult()
   Selection.Style = "result"
End Sub
```

Im Fall von XML-Code (den Sie hauptsächlich in Kapitel 8 finden) haben die Kommentare die folgende Form:

```
<!-- 08\Menüband_Button.xlsm -->
```

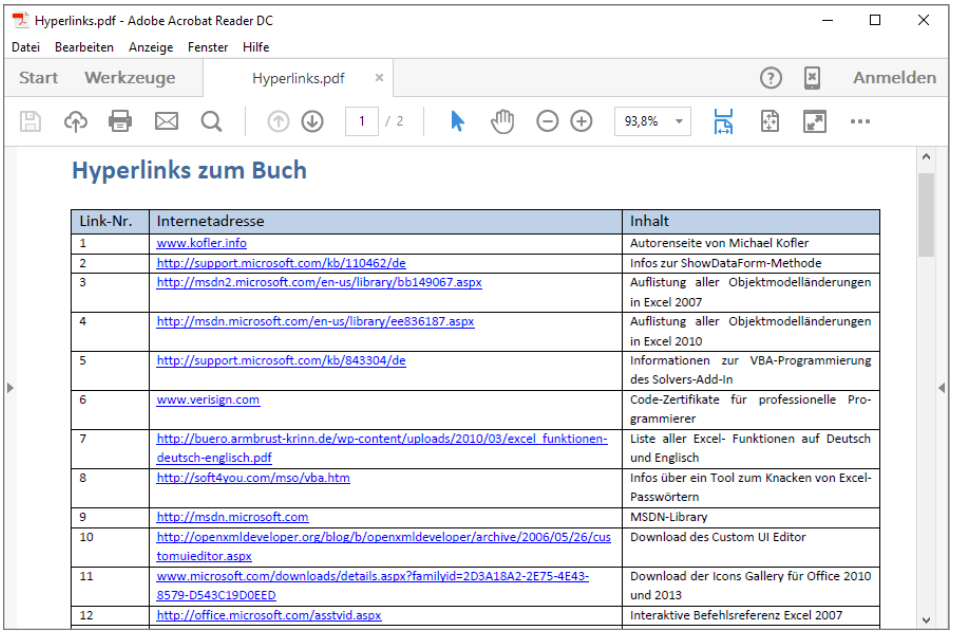
Kommentare in JavaScript-Dateien (Kapitel 15.9) schließlich sehen so aus:

```
/* 15\OfficeApps\SimpleApp\ComplexApp.js */
```

Internetadressen (Hyperlinks.pdf)

Der Text dieses Buchs enthält zahlreiche Verweise auf Internetadressen, wo Sie weiterführende Informationen finden, Tools herunterladen können etc. Da viele dieser „Links“ zu kompliziert sind, um sie abzutippen, haben wir sie in einem PDF-Dokument zusammengefasst. Es trägt den Namen *Hyperlinks.pdf* (siehe nächste Seite) und ist ebenfalls bei den Download-Dateien zum Buch im Ordner *Info* zu finden.

Die Links in diesem Dokument sind jeweils mit einer Nummer gekennzeichnet, die Sie auch im Buchtext in der Form *[Link x]* finden, wobei „x“ für die konkrete Link-Nummer steht. Zum Öffnen eines Links genügt ein Mausklick. Beim ersten Mal müssen Sie Ihrem Reader-Programm unter Umständen die Erlaubnis dazu erteilen.



Hyperlinks zum Buch

Link-Nr.	Internetadresse	Inhalt
1	www.kofler.info	Autorenseite von Michael Kofler
2	http://support.microsoft.com/kb/110462/de	Infos zur ShowDataForm-Methode
3	http://msdn2.microsoft.com/en-us/library/bb149067.aspx	Auflistung aller Objektmodelländerungen in Excel 2007
4	http://msdn.microsoft.com/en-us/library/ee836187.aspx	Auflistung aller Objektmodelländerungen in Excel 2010
5	http://support.microsoft.com/kb/843304/de	Informationen zur VBA-Programmierung des Solvers-Add-In
6	www.verisign.com	Code-Zertifikate für professionelle Programmierer
7	http://buero.armbrust-krinn.de/wp-content/uploads/2010/03/excel_funktionen-deutsch-englisch.pdf	Liste aller Excel- Funktionen auf Deutsch und Englisch
8	http://soft4you.com/mso/vba.htm	Infos über ein Tool zum Knacken von Excel-Passwörtern
9	http://msdn.microsoft.com	MSDN-Library
10	http://openxmldeveloper.org/blog/b/openxmldeveloper/archive/2006/05/26/customuieditor.aspx	Download des Custom UI Editor
11	www.microsoft.com/downloads/details.aspx?familyid=2D3A18A2-2E75-4E43-8579-D543C19D0EED	Download der Icons Gallery für Office 2010 und 2013
12	http://office.microsoft.com/asstvid.aspx	Interaktive Befehlsreferenz Excel 2007

Die Internetadressen in der Datei *Hyperlinks.pdf* können Sie direkt per Mausklick öffnen.

Und eine Entschuldigung

Wir sind uns bewusst, dass unter den Lesern dieses Buchs auch zahlreiche Frauen sind. Dennoch ist in diesem Buch immer wieder von *dem Anwender* die Rede, wenn wir keine geschlechtsneutrale Formulierung gefunden haben. Wir bitten dafür alle Leserinnen ausdrücklich um Entschuldigung. Wir sind uns des Problems bewusst, empfinden Doppelgleisigkeiten der Form *der/die Anwender/in* oder kurz *AnwenderIn* aber sprachlich als nicht schön – und zwar sowohl beim Schreiben als auch beim Lesen.

VBA-Elemente für die Programmierung von SmartArt-Diagrammen	
Description	Beschreibung eines SmartArt-Layouts
ID	Kennung eines SmartArt-Layouts
Larger	vergrößert den Diagrammknoten
msoSmartArt	Konstante für die Kennzeichnung eines SmartArt-Diagramms
Name	Name eines SmartArt-Layouts
Nodes	Auflistung aller Knoten in einem SmartArt-Diagramm oder übergeordneten Knoten
Promote	stuft den Diagrammknoten herauf
ReorderDown	verschiebt den Diagrammknoten nach unten
ReorderUp	verschiebt den Diagrammknoten nach oben
SmartArt	verweist auf das SmartArt-Diagramm in einem Shape-Objekt
SmartArtLayout	verweist auf ein einzelnes SmartArt-Layout
SmartArtLayouts	Auflistung aller Layouts für SmartArt-Diagramme
SmartArtNode	verweist auf einen einzelnen Knoten innerhalb der Nodes-Auflistung
TextFrame2.TextRange.Text	enthält die Beschriftung eines Diagrammknotens

10.8 Neue Diagrammtypen in Excel 2016

In Excel 2016 stehen dem Anwender nun die sechs neuen Diagrammtypen *Wasserfall*, *Histogramm*, *Pareto*, *Kastengrafik*, *Treemap* und *Sunburst* zur Verfügung. Während die ersten vier sich insbesondere für die Visualisierung (und Analyse) von finanziellen und statistischen Daten eignen, stehen mit Treemap und Sunburst wahre Spezialisten für die übersichtliche Darstellung von hierarchisch strukturierten Zahlentabellen zur Verfügung.

10.8.1 Programmierung von Wasserfall-Diagrammen

Ein Wasserfall-Diagramm verschafft dem Anwender ein klares Bild darüber, wie sich einzelne Faktoren auf ihre Nachfolger in einer Zahlenreihe auswirken. So kann man mit seiner Hilfe beispielsweise sehr gut darstellen, wie viel vom Umsatz eines Unternehmens nach Abzug diverser Kosten noch als Gewinn übrigbleibt. Bei einem solchen Einsatzszenario ergibt sich eine Reihe von treppenartig abfallenden Balken, die – mit etwas Fantasie – das Bild eines Wasserfalls ergeben.

Eine idealtypische Zahlentabelle für den Einsatz eines Wasserfall-Diagramms könnte wie folgt aussehen:

Bilanz	
Umsatz	**61.310,00 €**
Lohnkosten	- 15.056,00 €
Materialkosten	- 8.567,00 €
Transportkosten	- 5.824,45 €
Brutto	**31.862,55 €**

Sie finden diese Zahlentabelle in der Beispieldatei *10\Diagrammtypen2016.xlsm* auf einem Arbeitsblatt, das Sie mit einem Klick auf den Menü-Button „Wasserfall“ aktivieren können. Die Zahlentabelle ist mit dem Namen „rngDataWasserfall“ hinterlegt.

Das programmierte Anlegen eines Wasserfall-Diagramms beginnt nun damit, dass der benannte Arbeitsblattbereich mit zwei Befehlszeilen selektiert wird:

```
Set rngData = Range("rngDataWasserfall")
rngData.Select
```

Anschließend genügt die folgende Anweisung, um ein neues (eingebettetes) Diagramm vom Typ Wasserfall (*xlWaterfall*) zu generieren und ihm den selektierten Arbeitsblattbereich als Datenquelle zuzuweisen:

```
Set shpDiagram = ActiveSheet.Shapes.AddChart2(-1, xlWaterfall)
```

Hinweis

Die Programmzeile generiert das Objekt *shpDiagram*, das wie jedes eingebettete (d. h. in ein Arbeitsblatt eingefügte) Diagramm vom Typ *Shape* ist. Das eigentliche Diagramm darin ist über die *Chart*-Eigenschaft des Shape-Objekts zugänglich. Hinweis im Hinweis: Alle Shape-Objekte eines Arbeitsblatts sind über dessen *Shapes*-Auflistung abrufbar.

Das neu angelegte Diagramm hat jedoch noch wenig Ähnlichkeit mit einem Wasserfall, da sein letzter Balken – er stellt den errechneten „Brutto“-Wert am Ende der Zahlentabelle dar – auf der gleichen Höhe wie der Balken „Transportkosten“ beginnt und damit „nach oben“ statt „nach unten“ geht.

Um dieses Problem zu lösen, muss man Excel mitteilen, dass es sich bei diesem letzten Wert nicht um einen Einzelwert handelt, der von seinem Vorläufer abgezogen oder zu diesem hinzuaddiert wird, sondern um eine Summe. Händisch würde man das realisieren, indem man auf den letzten Diagrammbalken doppelklickt und dann das Kontrollkästchen „Als Summe festlegen“ am rechten Bildschirmrand einschaltet.

Die programmierte Variante der Lösung ist kaum schwieriger. Sie beginnt mit einem Verweis auf den letzten Datenpunkt des Diagramms, der sich wie folgt herstellen lässt:

```
Set objLastPoint = _
    shpDiagram.Chart.SeriesCollection(1).Points( _
    shpDiagram.Chart.SeriesCollection(1).Points.Count)
```

Danach muss man nur noch die *IsTotal*-Eigenschaft des *objLastPoint*-Objekts (das vom Typ *Point* ist) auf *True* setzen:

```
objLastPoint.IsTotal = True
```

Das Ergebnis dieser Aktion sollte dann so aussehen wie die nachfolgende Abbildung.

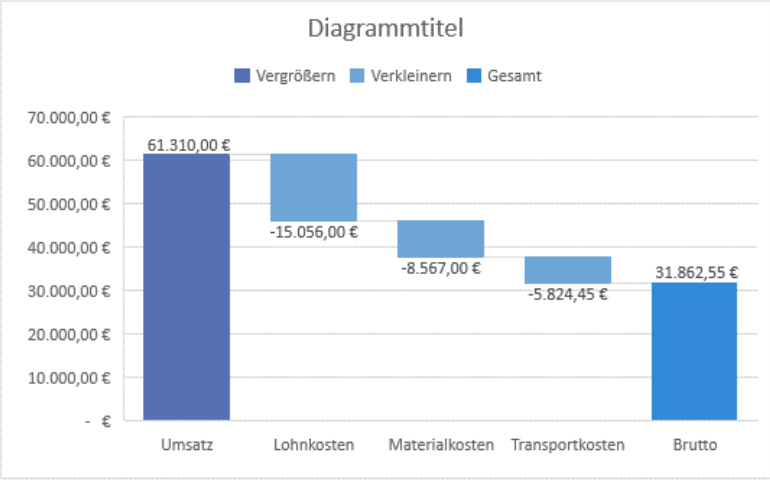


BILD 10.20: Das programmierte Wasserfall-Diagramm zeigt anschaulich, wie viel vom Umsatz nach Abzug diverser Kosten noch als (Brutto-)Gewinn übrigbleibt.

10.8.2 Programmierung von Histogrammen

Ein Histogramm ist eine grafische Darstellung der Häufigkeitsverteilung bestimmter Merkmale. Es erfordert die Einteilung der Daten in Klassen, die eine konstante oder variable Breite haben können. In Excel bestehen Histogramme aus nebeneinanderliegenden Balken, die jeweils einer Klasse entsprechen. Die Höhe eines jeden Balkens stellt die sogenannte Häufigkeitsdichte der betreffenden Klasse dar.

Was ziemlich kompliziert und wissenschaftlich klingt, lässt sich im Excel-Alltag aber sehr einfach und gewinnbringend anwenden. So könnte ein Lehrer beispielsweise mit Hilfe eines Histogramms sehr leicht feststellen, wie sich die Leistungen seiner Schüler verteilen. Als Datenbasis einer solchen Analyse könnte eine Zahlentabelle dienen, die in Spalte A die Namen der Schüler und dahinter in Spalte B deren Noten (oder Zensuren) enthält:

Schüler	Note
Schüler 1	3
Schüler 2	3
Schüler 3	6
Schüler 4	6
Schüler 5	5
Schüler 6	6
Schü	6

Die Abbildung oben zeigt nur einen Ausschnitt einer solchen Zahlentabelle; das vollständige Exemplar finden Sie in der Beispieldatei *10\Diagrammtypen2016.xlsm* auf einem Arbeitsblatt, das Sie mit einem Klick auf den Menü-Button „Histogramm“ aktivieren können.

Der Inhalt der Tabelle ist mit dem Namen „rngDataHistogramm“ hinterlegt, der im ersten Schritt der Programmierung wie folgt selektiert wird:

```
Set rngData = Range("rngDataHistogramm")
rngData.Select
```

Die Selektion stellt nun automatisch den Datenbereich des Histogramms dar, das mit der folgenden Programmzeile generiert wird:

```
Set shpDiagram = ActiveSheet.Shapes.AddChart2(-1, xlHistogram)
```

Das resultierende Diagramm (siehe nachfolgende Abbildung) stellt die Verteilung der Schulnoten in drei Klassen dar, die man mit „Gut“, „Mittel“ und „Schlecht“ überschreiben könnte. Der Lehrer sieht daran sofort, dass die Mehrzahl seiner Schüler der Klasse „Schlecht“ angehören und deren Versetzung somit gefährdet ist.

Sollte sich der Lehrer eine feinere Klassenbildung wünschen – beispielsweise für jede Schulnote einen eigenen Balken – so kann er die Eigenschaften des Histogramms manuell ändern. Dazu doppelklickt er auf die untere Diagrammachse und ändert dann im Dialogfeld *Achsenoptionen* am rechten Bildschirmrand die Einstellung *Anzahl der Container* (das sind die Klassen des Histogramms) auf „6“ und die Einstellung *Containerbreite* auf „0,999“. Ein programmierter Zugriff auf diese Einstellungen ist leider nicht möglich, da diese offenbar keinen Eingang in das Objektmodell von Excel 2016 gefunden haben.

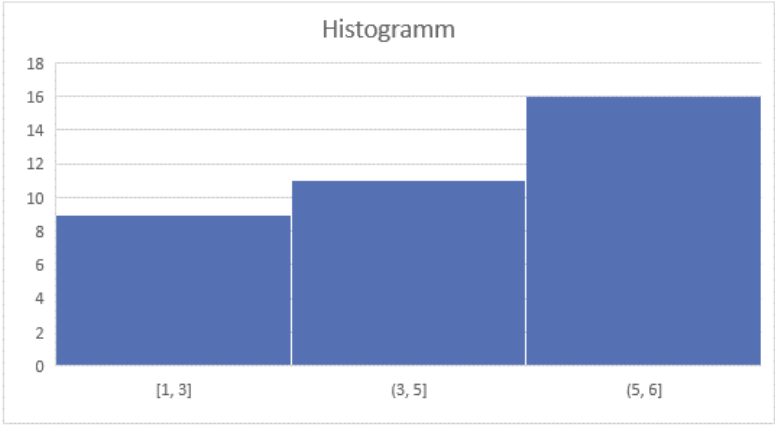


BILD 10.21: Ein Histogramm fasst Merkmale (hier Schulnoten) zu Klassen zusammen und stellt dann deren Häufigkeiten als Balken dar.

10.8.3 Programmierung von Pareto-Diagrammen

Ein Pareto-Diagramm ist die Sonderform eines Säulendiagramms, bei dem die einzelnen Werte der Größe nach geordnet angezeigt werden. Dabei befindet sich der größte Wert ganz links, der kleinste ganz rechts. Neben dem Säulendiagramm enthält ein Pareto-Diagramm aber auch noch ein Liniendiagramm, das die prozentualen Anteile der Einzelwerte am Gesamtergebnis in akkumulierter Form anzeigt.

Auch das klingt komplizierter als es ist. In der Praxis benutzt man ein Pareto-Diagramm immer dann, wenn man die wichtigsten Größen für das Zustandekommen eines Ergebnisses erkennen und sie von den weniger wichtigen Größen unterscheiden möchte. Der Chef eines Versicherungsunternehmens könnte ein Pareto-Diagramm beispielsweise dazu verwenden, die Top-Drei seiner Außendienstmitarbeiter zu ermitteln. Dazu schreibt er die Namen seiner Mitarbeiter in die erste Spalte einer Zahlentabelle, die Umsätze in die zweite. Das Ergebnis könnte dann wie folgt aussehen:

Vertreter	Umsatz
Vertreter 1	11.445,00 €
Vertreter 2	11.829,00 €
Vertreter 3	14.547,00 €
Vertreter 4	14.702,00 €
Vertreter 5	10.203,00 €
Vertreter 6	10.669,00 €
Vertreter 7	10.107,00 €
Vertreter 8	31.467,00 €
Vertreter 9	35.150,00 €
Vertreter 10	30.064,00 €

Sie finden diese Tabelle, wenn Sie die Beispieldatei *10\Diagrammtypen2016.xlsm* öffnen und auf den Menü-Button „Pareto“ klicken.

Das programmierte Anlegen des Pareto-Diagramms beginnt nun mit zwei Programmzeilen, die den Inhalt der Zahlentabelle selektieren. Dazu wurde dieser zuvor mit dem Namen „rngDataPareto“ gekennzeichnet:

```
Set rngData = Range("rngDataPareto")
rngData.Select
```

Die Zeilen machen die selektierten Zellen zum Datenbereich des neuen Diagramms, das dann mit dem folgenden Einzeiler generiert wird:

```
Set shpDiagram = ActiveSheet.Shapes.AddChart2(-1, xlPareto)
```

Über die *Chart*-Eigenschaft des Objekts *shpDiagram* können Sie anschließend auf alle Standardeigenschaften des Diagramms zugreifen, beispielsweise um den Titel des Diagramms nach Ihren Wünschen zu ändern:

```
shpDiagram.Chart.ChartTitle.Text = "Mein Pareto-Diagramm"
```

Spezielle Eigenschaften eines Pareto-Diagramms haben leider keinen Einzug in Excels Objektmodell gefunden.

Das Resultat der *AddChart2*-Anweisung oben sieht dann so aus wie in der nachfolgenden Abbildung. Das Diagramm verrät dem Chef mit einem Blick, dass seine drei besten Mitarbeiter – sie haben die (Personal-)Nummern 9, 8 und 10 – mehr als die Hälfte des Umsatzes generieren. Zeit für eine Lohnerhöhung!

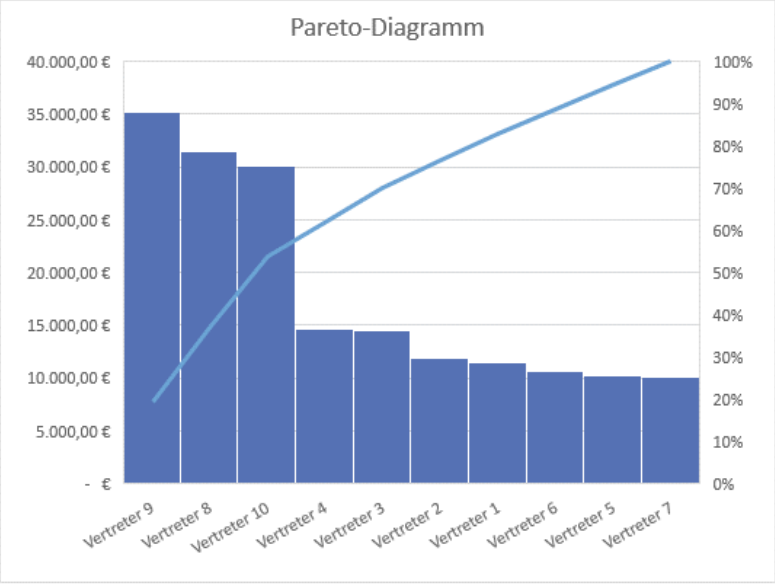


BILD 10.22: Wer wissen möchte, welche Faktoren den größten Anteil an einem bestimmten Ergebnis haben, verwendet ein Pareto-Diagramm. Hier sieht man, dass drei Mitarbeiter mehr als die Hälfte des Umsatzes generieren.

10.8.4 Programmierung von Kastengrafik-Diagrammen

Bei einem Kastengrafik-Diagramm, das im Englischen auch Boxplot oder Box-Whisker-Plot genannt wird, steht ähnlich wie bei einem Histogramm (siehe Abschnitt 10.8.2) die Verteilung von Daten im Mittelpunkt. Allerdings geht das Kastengrafik-Diagramm weit über die Anzeige schlichter Häufigkeiten hinaus, indem es für jede Datenreihe eine komplexe Grafik generiert, die verschiedene Streuungs- und Lagemaße in einer Darstellung kombiniert. Zentrales (und namensgebendes) Element einer Kastengrafik ist ein Rechteck. Es umfasst den Zahlenbereich, in dem die mittleren 50 Prozent der Daten liegen. Die beiden Linien ober- und unterhalb des Rechtecks werden Antennen genannt. Sie repräsentieren die Spannweite der oberen und unteren 25 Prozent der Daten und markieren an ihrem Ende den größten und kleinsten Wert der Datenreihe. Innerhalb des Rechtecks zeigt Excel den Median – das ist der mittlere Wert in der sortierten Datenreihe – mit einer Querlinie an; ein Kreuz markiert das arithmetische Mittel.

Mit Hilfe eines Kastengrafik-Diagramms könnte sich ein Buchhändler beispielsweise einen detaillierten Einblick verschaffen, wie sich die Umsätze verschiedener Buchgenres gestalten. Dazu legt er zunächst eine Zahlentabelle an, die in Spalte 1 den Titel, in Spalte 2 das Genre und in der letzten Spalte schließlich den Preis des jeweiligen Buchs verzeichnet. Die

nachfolgende Abbildung zeigt nur einen Ausschnitt; die vollständige Tabelle finden Sie, wenn Sie die Beispieldatei *10\Diagrammtypen2016.xlsm* öffnen und auf den Menü-Button „Kastengrafik“ klicken.

Buch	Genre	Preis
Buch 1	Fantasy	24,00 €
Buch 2	Fantasy	16,00 €
Buch 3	Satire	24,00 €
Buch 4	Satire	28,00 €
Buch 5	Fantasy	13,00 €
Buch 6	Satire	21,00 €
Buch 7	Belletristik	42,00 €
Buch 8		

Die letzten beiden Spalten der Tabelle bilden den Datenbereich des anzulegenden Diagramms, wozu sie mit dem Namen „rngDataKasten“ hinterlegt wurden. Dieser benannte Bereich wird nun im ersten Schritt der Programmierung selektiert:

```
Set rngData = Range("rngDataKasten")
rngData.Select
```

Die folgende Anweisung erstellt nun auf der Grundlage der Selektion ein neues Diagramm vom Typ Kastengrafik:

```
Set shpDiagram = ActiveSheet.Shapes.AddChart2(-1, xlBoxwhisker)
```

Das Ergebnis dieser Aktion sollte der nachfolgenden Abbildung entsprechen. Über die *Chart*-Eigenschaft des Objekts *shpDiagram* kann der Entwickler wieder auf alle Standardeigenschaften des Diagramms zugreifen. Wer die spezifischen Eigenschaften eines Kastengrafik-Diagramms ändern möchte, muss das manuell über die Formatierungsdialoge von Excel erledigen. In Excels Objektmodell finden sich keine entsprechenden Erweiterungen.

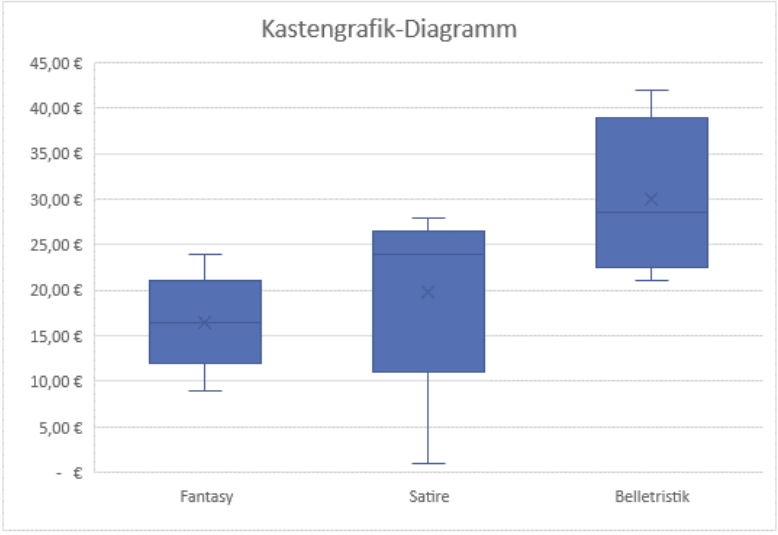


BILD 10.23: Kastengrafik-Diagramme enthalten diverse statistische Informationen darüber, wie sich die Daten einer Reihe – hier geht es um die Umsatzzahlen innerhalb von Buchgenres – verteilen.

10.8.5 Programmierung von Treemap-Diagrammen

Ein Treemap-Diagramm (auch Baumkarte genannt) visualisiert die Größenverhältnisse innerhalb von hierarchisch strukturierten Daten. Dazu stellt es Rechtecke innerhalb von Rechtecken dar. Die Flächen dieser Rechtecke sind dabei stets proportional zur Größe der darzustellenden Dateneinheit.

Ein typisches Einsatzszenario für ein Treemap-Diagramm könnte so aussehen: Ein Restaurantbesitzer möchte einen vollständigen Überblick über die Umsatzverteilung seines Unternehmens. Dazu möchte er nicht nur wissen, wie sich die Umsätze über die drei Mahlzeitenangebote Frühstück, Mittag- und Abendessen verteilen, sondern auch innerhalb der einzelnen Mahlzeitenangebote über die angebotenen Speisen und Getränke. Zu diesem Zweck erstellt er die folgende Zahlentabelle, die Sie in der Beispieldatei *10\Diagrammtypen2016.xlsm* nach einem Klick auf den „Treemap"-Button finden:

Mahlzeit	Speiseart	Name	Umsatz
Frühstück	Getränk	Milch	6,00 €
		Kaffee	9,00 €
		Orangensaft	7,00 €
	Speise	Brötchen	32,00 €
		Marmelade	11,00 €
Mittagessen	Getränk	Milchshake	15,00 €
		Smoothy	17,00 €
	Speise	Schnitzel	46,00 €
		Kartoffeln	35,00 €
Abendessen	Getränk	Tee	9,00 €
		Wein	44,00 €
	Speise	Steak	99,00 €
		Fisch	43,00 €
		Auflauf	33,00 €

Diese Zahlentabelle ist mit dem Namen „rngDataTreemap" hinterlegt. Die folgenden Anweisungen

```
Set rngData = Range("rngDataTreemap")
rngData.Select
```

selektieren den benannten Bereich und machen ihn automatisch zur Datenbasis des neuen Treemap-Diagramms, das nach Ausführung der Code-Zeile

```
Set shpDiagram = ActiveSheet.Shapes.AddChart2(-1, xlTreemap)
```

entsteht. Das neu angelegte Diagramm entspricht jedoch der Standard-Diagrammvorlage, die weniger übersichtlich gestaltet ist als eine andere Vorlage, auf die man mit folgender Anweisung umschalten kann:

```
shpDiagram.Chart.ChartStyle = 413
```

Jetzt erscheinen die Namen der Mahlzeitenangebote als graue Balken direkt oberhalb der Rechtecke, die die Umsätze der zugehörigen Speisen und Getränke visualisieren. Da die Diagrammlegende jetzt überflüssig ist, schalten wir sie mit dieser Code-Zeile ab:

```
shpDiagram.Chart.HasLegend = False
```

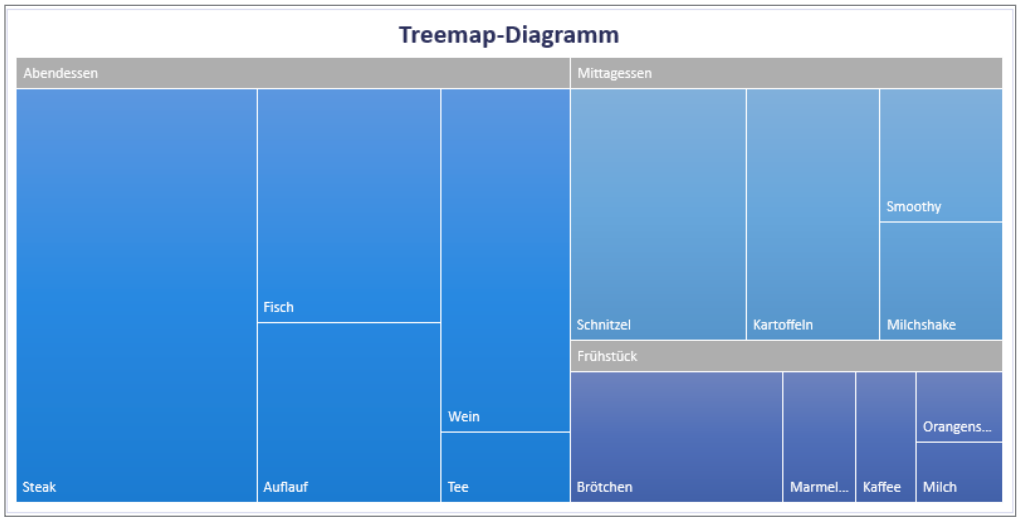


BILD 10.24: Dieses Treemap-Diagramm verrät dem Restaurantbesitzer, wie sich die Umsätze seines Unternehmens über die verschiedenen Mahlzeiten und die jeweils dabei angebotenen Speisen und Getränke verteilen.

10.8.6 DirectoryMap – Inhaltsverzeichnisse visualisieren

Irgendwann kommt jede Festplatte an ihre Kapazitätsgrenzen. Wer dann effektiv Platz für neue Inhalte schaffen will, sollte beim Aufräumen mit den größten Platzverschwendern beginnen. Die aufzuspüren, ist jedoch nicht immer leicht, da der Windows-Explorer nur über den Umweg des Eigenschaften-Dialogs Auskunft über Verzeichnisgrößen liefert. Wenige Zeilen VBA und der neue Diagrammtyp Treemap (siehe Abschnitt 10.8.5) genügen allerdings, um das Manko zu beheben.

Inside DirectoryMap.xlsm

Die Beispieldatei *10\DirectoryMap.xlsm* enthält nur ein einziges Arbeitsblatt namens „Directory“. An dessen Spitze befindet sich ein Shape mit Button-Optik, das beim Anklicken das Makro *ShowDirectoryMap* aufruft. Das bringt zunächst das typische Dialogfeld zur Auswahl eines Ordners auf den Bildschirm. Dazu bedient sich das Makro im Rahmen der ActiveX-Automation (siehe Abschnitt 15.6) des Windows-eigenen *Shell*-Objekts, das es mit der Zeile

```
Set objShell = CreateObject("Shell.Application")
```

einbindet. Die Darstellung des Dialogs erledigt dann die *BrowseForFolder*-Methode des Shell-Objekts, das den ausgewählten Ordner als *Folder*-Objekt zurückgibt:

```
Set objFolder = objShell.BrowseForFolder(0, strPrompt, &H8, 0)
```

Das aufrufende Makro liest den Pfad des gewählten Ordners aus *objFolder.Items.Item.Path* und speichert ihn in der Variablen *strPath*.

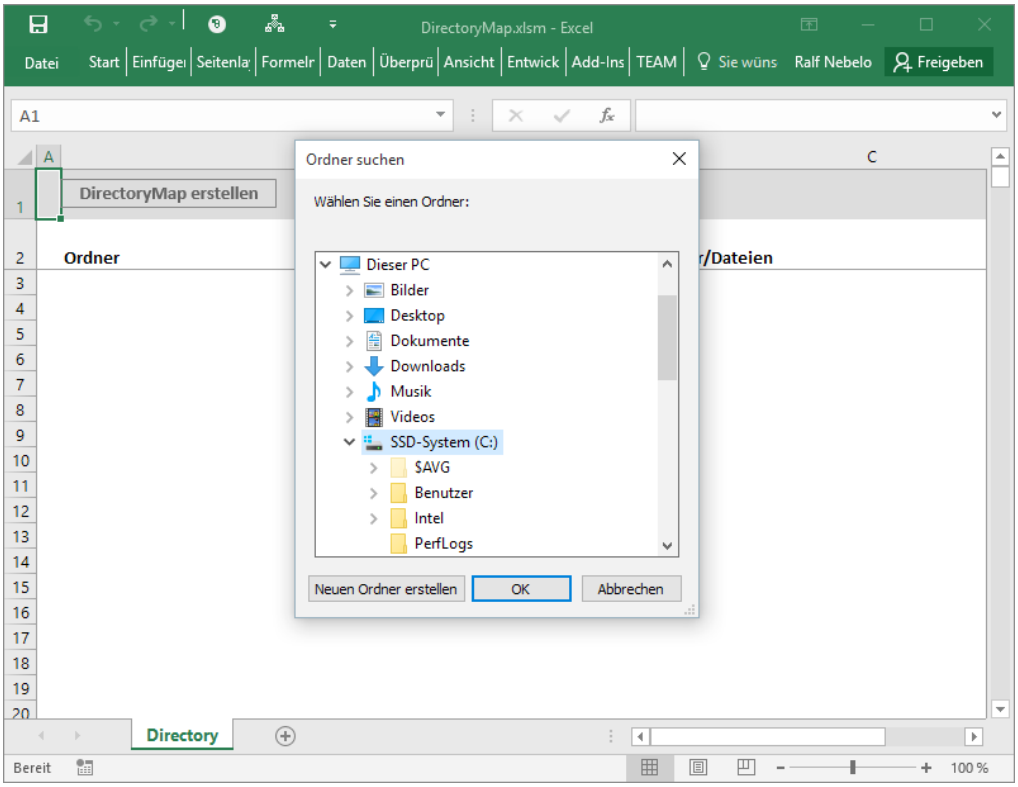


BILD 10.25: Ein Klick auf den (Shape-)Button „DirectoryMap erstellen“ bringt zunächst ein Dialogfeld hervor, das die Auswahl des gewünschten Ordners oder Laufwerks erlaubt.

Da das Makro mit sehr ausführlichen Kommentaren versehen ist, können wir uns im Folgenden auf die wichtigsten Aspekte der Programmierung beschränken. Auf die *CreateDirectory*-Funktion beispielsweise, die das Inhaltsverzeichnis des ausgewählten Ordners generiert und es als Datenbasis für das spätere Treemap-Diagramm in das Arbeitsblatt „Directory“ schreibt.

Inhaltsverzeichnis generieren

Da VBA nur über unzureichende Bordmittel für den Zugriff auf das Dateisystem verfügt, greift es diesbezüglich – und wiederum im Rahmen der ActiveX-Automation – auf die Dienste der *File System Objects* (siehe Abschnitt 5.6.1) zurück, die es wie folgt einbindet:

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
```

Mit Hilfe der *GetFolder*-Methode der File System Objects verschafft sich das Makro einen Verweis auf den ausgewählten Ordner, den es in der Variablen *objDir* entgegennimmt:

```
Set objDir = objFSO.GetFolder(strPath)
```

Anschließend durchläuft es per *For-Each-Next*-Schleife alle Unterordner des Ordners und schreibt deren Namen in die erste Spalte der Inhaltsverzeichnis-Tabelle:

```
For Each objSubDir In objDir.SubFolders
    wksSheet.Cells(lngRow, lngFirstColumn).Value = objSubDir.Name
    …
Next
```

Innerhalb dieser Schleife gibt es zwei weitere (eingeschlossene) Schleifen, die die Inhalte des jeweiligen Unterordners durchlaufen und die Namen und Größen (in Bytes) der dabei gefundenen (Unter-Unter-)Ordner und Dateien in die zweite und dritte Spalte der Inhaltsverzeichnis-Tabelle ausgeben:

```
For Each objSubDir2 In objSubDir.SubFolders
    wksSheet.Cells(lngRow, lngFirstColumn + 1).Value = _
        objSubDir2.Name
    wksSheet.Cells(lngRow, lngFirstColumn + 2).Value = _
        objSubDir2.Size
    lngRow = lngRow + 1
Next

For Each objFile In objSubDir.Files
    wksSheet.Cells(lngRow, lngFirstColumn + 1).Value = _
        objFile.Name
    wksSheet.Cells(lngRow, lngFirstColumn + 2).Value = _
        objFile.Size
    lngRow = lngRow + 1
Next
```

Zum Abschluss ihrer Tätigkeit merkt sich die *CreateDirectory*-Funktion den Bereich (Range) der soeben erstellten Inhaltsverzeichnis-Tabelle ...

```
Set rngData = Range(wksSheet.Cells(lngFirstRow, _
    lngFirstColumn), wksSheet.Cells(lngRow - 1, lngFirstColumn + 2))
```

... und gibt ihn mit der folgenden Zeile an das aufrufende Makro zurück:

```
Set CreateDirectory = rngData
```

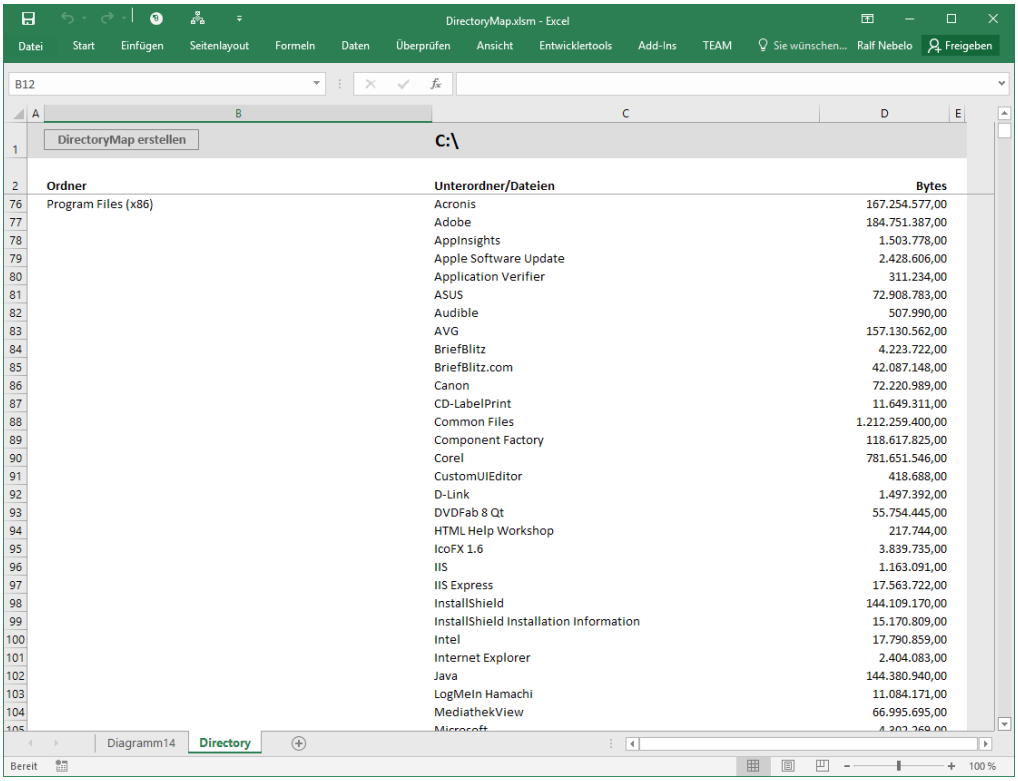


BILD 10.26: Das Makro generiert eine Verzeichnis-Tabelle, die alle Ordner des gewählten Verzeichnisses und deren jeweilige Unterordner und Dateien auflistet.

Verzeichnisinhalte per Treemap darstellen

Nachdem dem Makro nun der Bereich des generierten Inhaltsverzeichnisses bekannt ist, muss es diesen nur noch zur Datenbasis eines neuen Treemap-Diagramms machen. Das Anlegen des Diagramms beginnt mit der Auswahl der Inhaltsverzeichnis-Tabelle:

```
rngData.Select
```

Die folgende Zeile erstellt dann ein neues Treemap-Diagramm und weist diesem die Diagrammvorlage mit der internen Nummer 413 zu, welche die Ordnernamen zwecks besserer Übersicht als graue Balken über den jeweiligen Ordnerinhalten erscheinen lässt:

```
Set shpDiagram = wksDirectory.Shapes.AddChart2(413, xlTreemap)
```

Jetzt müssen in dem neuen Diagramm nur noch die überflüssige Legende abgeschaltet und die Überschrift so geändert werden, dass diese den Pfad des ausgewählten Ordners zeigt:

```
With shpDiagram.Chart
    .HasLegend = False
    .ChartTitle.Text = "Inhalt von " & strPath
End With
```

Die finale Anweisung …

```
shpDiagram.Chart.Location xlLocationAsNewSheet
```

… verschiebt das Diagramm(-Shape) schließlich aus dem Arbeitsblatt in ein eigenständiges Diagrammblatt, wo es mehr Platz für eine detaillierte Darstellung der Verzeichnisinhalte bekommt.

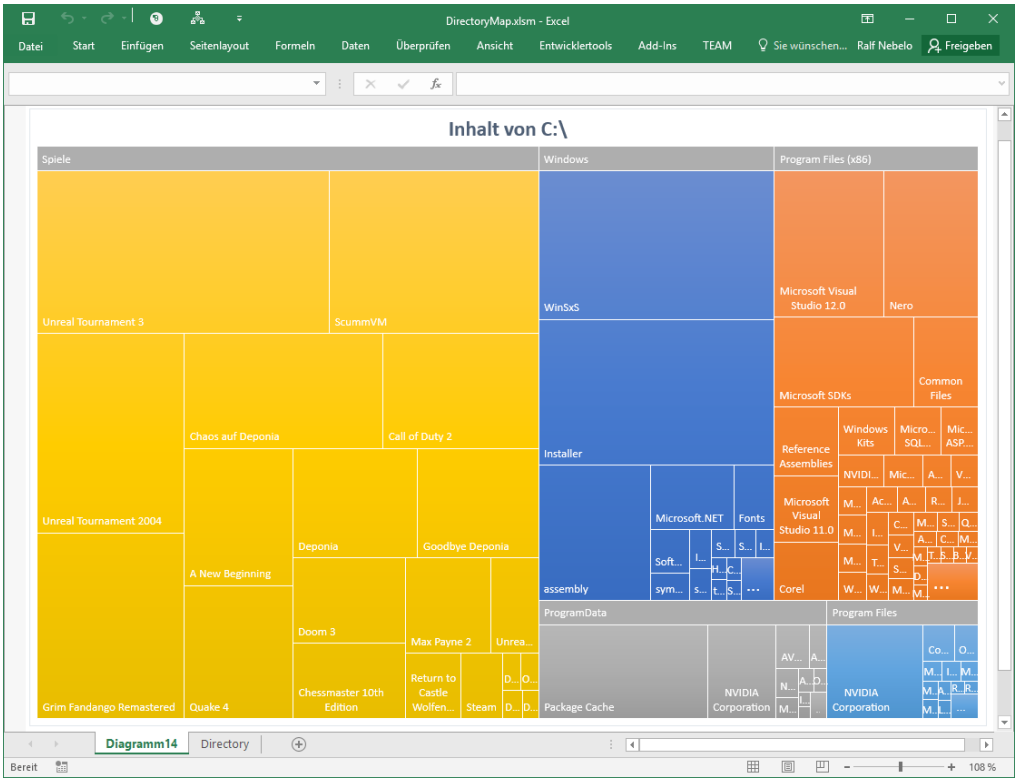


BILD 10.27: Das Treemap-Diagramm stellt die Ordnergrößen auf der Festplatte (und die offensichtliche Vorliebe des Autors für Computerspiele ;-) eindrücklich dar.

10.8.7 Programmierung von Sunburst-Diagrammen

Ähnlich dem Treemap-Diagramm (siehe Abschnitt 10.8.5) stellt auch das Sunburst-Diagramm die Größenverhältnisse innerhalb von hierarchisch strukturierten Daten dar. Allerdings greift es dazu nicht auf verschachtelte Rechtecke zurück, sondern auf ebenso verschachtelte Ringe oder Kreise, welche die einzelnen Hierarchiestufen symbolisieren. Dabei stellt der innerste Kreis die höchste Hierarchiestufe dar, der äußerste die tiefste. Jeder Kreis ist in Segmente unterteilt, deren Flächen proportional den Größen der dargestellten Daten entsprechen.

Mit Hilfe eines Sunburst-Diagramms könnte ein großes Autohaus beispielsweise die Umsatzverteilung über drei Hierarchiestufen hinweg – nämlich Automarken, Kundentypen und Verkäufer – detailliert darstellen. Dazu braucht es zunächst eine Zahlentabelle wie die folgende, die in der Beispieldatei *10\Diagrammtypen2016.xlsm* nach einem Klick auf den „Sunburst"-Button zu finden ist:

Marke	Kundentyp	Verkäufer	Umsatz
Audi	Gewerbe		93
	Endkunden	Müller	32
		Meier	30
		Hansen	30
	Behörden		95
VW	Gewerbe		79
	Endkunden	Harms	14
		Seisner	13
		Engel	12
		Dressen	14
		Madler	14
	Behörden		67
Porsche	Gewerbe		48
	Endkunden	Schulz	21
		Hackel	30
	Behörden		57
Seat	Gewerbe		24
	Endkunden		25
	Behörden		34

Die Tabelle ist mit dem Bereichsnamen „rngDataSunburst" hinterlegt. Die Code-Zeilen

```
Set rngData = Range("rngDataSunburst")
rngData.Select
```

selektieren den benannten Bereich und machen ihn damit automatisch zum Datenbereich des anzulegenden Sunburst-Diagramms (siehe nachfolgende Abbildung), das mit der Anweisung

```
Set shpDiagram = ActiveSheet.Shapes.AddChart2(-1, xlSunburst)
```

generiert wird. Wie die meisten neuen Excel-2016-Diagramme verfügt das Sunburst-Diagramm über eine Reihe von spezifischen Eigenschaften, die der Anwender mangels Integration in das Objektmodell von Excel leider nur manuell mit Hilfe der einschlägigen Formatierungsdialoge verändern kann.

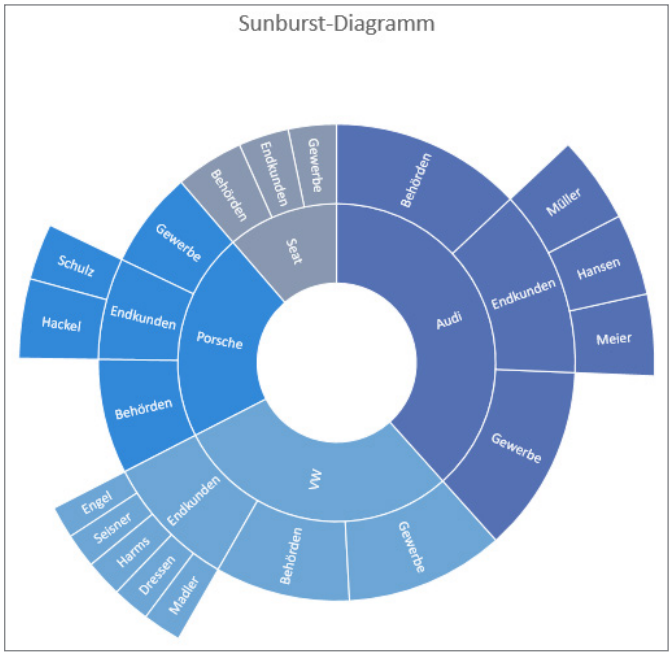


BILD 10.28:
Ein Sunburst-Diagramm stellt für jede Hierarchiestufe einen Kreis bereit. Dessen Segmente visualisieren die Größenverhältnisse der zugehörigen Daten.

10.9 Zeichnungsobjekte (Shapes)

Überblick

Das *Shape*-Objekt dient primär zur Darstellung von AutoFormen (Linien, Rechtecke, Pfeile, Sterne etc. – siehe Einfügen | Formen). Es löst damit die diversen Zeichnungsobjekte aus Excel 5/7 ab. Verwirrung stiftet allerdings die große Anzahl verwandter Objekte.

Hierarchie der Shape-Objekte

Worksheet/Chart	
└ *Shapes*	alle *Shape*-Objekte innerhalb des Blatts
└ *Shape*	ein *Shape*-Objekt
├ *ConnectorFormat*	Verbindung zu anderen Objekten
├ *ControlFormat*	zusätzliche Eigenschaften für Steuerelemente
├ *FillFormat*	Hintergrundmuster (via *Fill*-Eigenschaft)
├ *GroupShapes*	Einzelobjekte (via *GroupItems*, wenn *Type=msoGroup*)
│ └ *Shape*	
├ *HyperLink*	Querverweise und Internetlinks
└ *LineFormat*	Linieneigenschaften (via Line*)*

OLE, ActiveX-Automation	
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 allzu viele 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

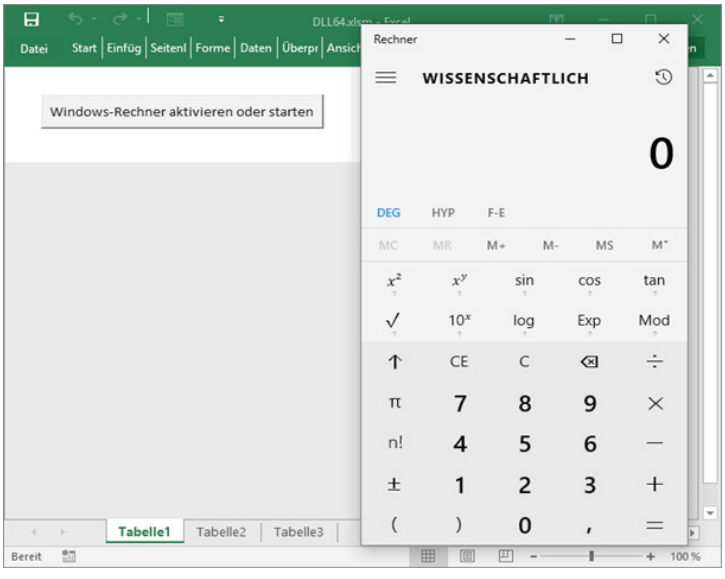


BILD 15.13: Das Beispielprogramm nutzt diverse API-Funktionen, die den Windows-Rechner je nach Zustand starten oder aktivieren.

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.

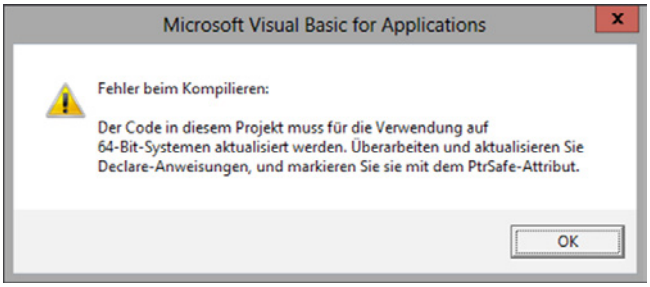


BILD 15.14: In der 64-Bit-Version von Excel müssen *Declare*-Anweisungen zwingend das Schlüsselwort *PtrSafe* enthalten, sonst gibt es Gemecker in Form dieser Fehlermeldung.

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 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
```

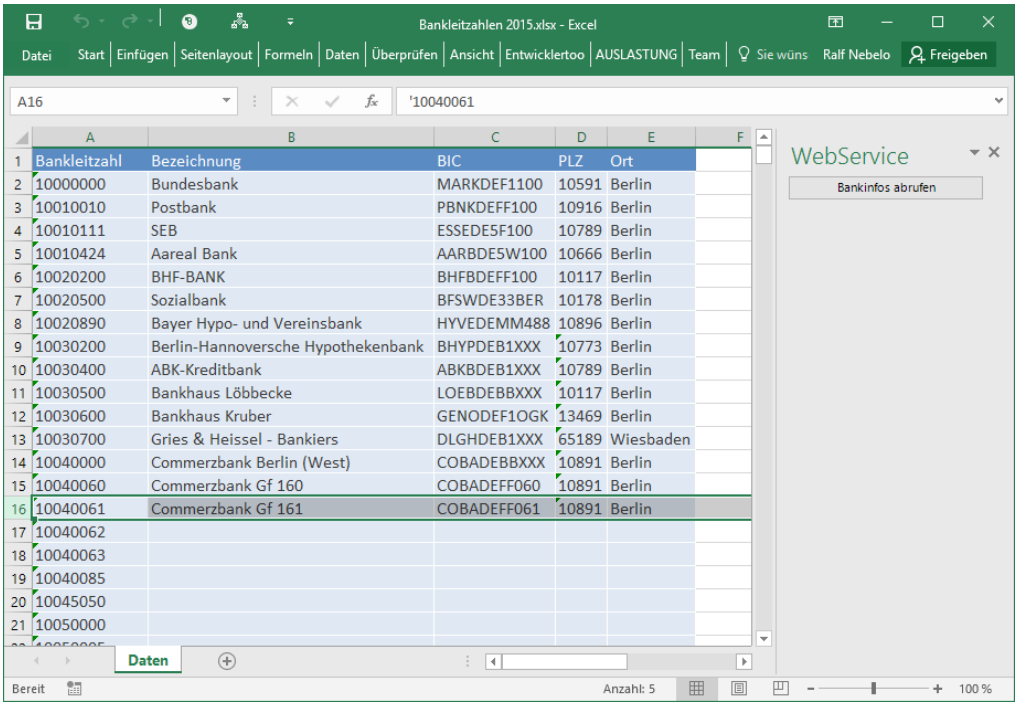
Hinweis

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 (und neuer) zur Verfügung steht
LongPtr	variabler Datentyp, der auf 32-Bit-Versionen von Excel 2010 (und neuer) 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 Anweisungsblöcken für VBA 7 und ältere VBA-Versionen ermöglicht

Mit einem Druck auf F5 führen Sie das Projekt aus. Die Aktion startet Excel mit einer neuen Arbeitsmappe und dem neuen Aufgabenbereich am rechten Rand des Programmfensters. Zum Ausprobieren tippen Sie Ihre ganz persönliche Bankleitzahl (ohne Leerzeichen!) in eine beliebige Zelle, markieren diese anschließend und klicken dann auf die Schaltfläche BANKINFOS ABRUFEN.



	A	B	C	D	E
1	Bankleitzahl	Bezeichnung	BIC	PLZ	Ort
2	10000000	Bundesbank	MARKDEF1100	10591	Berlin
3	10010010	Postbank	PBNKDEFF100	10916	Berlin
4	10010111	SEB	ESSEDE5F100	10789	Berlin
5	10010424	Aareal Bank	AARBDE5W100	10666	Berlin
6	10020200	BHF-BANK	BHFBDEFF100	10117	Berlin
7	10020500	Sozialbank	BFSWDE33BER	10178	Berlin
8	10020890	Bayer Hypo- und Vereinsbank	HYVEDEMM488	10896	Berlin
9	10030200	Berlin-Hannoversche Hypothekenbank	BHYPDEB1XXX	10773	Berlin
10	10030400	ABK-Kreditbank	ABKBDEB1XXX	10789	Berlin
11	10030500	Bankhaus Löbbecke	LOEBDEBBXXX	10117	Berlin
12	10030600	Bankhaus Kruber	GENODEF1OGK	13469	Berlin
13	10030700	Gries & Heissel - Bankiers	DLGHDEB1XXX	65189	Wiesbaden
14	10040000	Commerzbank Berlin (West)	COBADEBBXXX	10891	Berlin
15	10040060	Commerzbank Gf 160	COBADEFF060	10891	Berlin
16	10040061	Commerzbank Gf 161	COBADEFF061	10891	Berlin
17	10040062				
18	10040063				
19	10040085				
20	10045050				
21	10050000				

BILD 15.25: Das fertige VSTO-Projekt demonstriert, wie leicht sich der BIC und andere Bankdaten mit der Hilfe eines Webdienstes abrufen lassen. Dazu muss man diesem nur eine gültige Bankleitzahl übergeben.

15.9 Office-Add-ins

Hinter dem neuen Befehl MEINE ADD-INS im Menüband EINFÜGEN von Excel 2016 verbirgt sich ein mit Office 2013 eingeführtes und damals radikal neues Erweiterungskonzept, das sogenannte „Office-Add-ins“ (die in Office 2013 noch „Office-Apps“ hießen) an die Stelle von VBA-Makros setzen und Webtechniken und Cloud-Dienste direkt in die Bedienoberflächen aller Office-Anwendungen integrieren will.

Obwohl sich Office-Add-ins 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 Online-Shop, den man direkt aus den Office-Anwendungen heraus erreicht.

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

15.9.1 Bestandteile eines Office-Add-ins

Ein Office-Add-in ist im Grunde eine normale Webanwendung, die aber nicht auf einem Server im Internet, sondern in Office „gehostet" und auch angezeigt wird. Das typische Office-Add-in 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 des Office-Add-ins.
- **Skriptdatei:** Das ist zumeist eine JavaScript-Datei, die den Programmcode des Add-ins 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 eines Office-Add-ins 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 ein Office-Add-in 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 des Office-Add-ins, 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 das Office-Add-in bereit und weist allen Bestandteilen der Webseite die Office-typischen Schriftarten und Formatierungen zu.
- **Manifestdatei:** Diese XML-Datei ist das Bindeglied zwischen den einzelnen Add-in-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 des Add-ins, seine 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 Add-in-Webseite her. Die Bibliothek sorgt dafür, dass das Add-in 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 Add-in 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 Add-in-Elemente lässt bereits erahnen, dass die Entwicklung von Office-Add-ins 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-Add-in-Entwickler *[Link 44]*:

https://msdn.microsoft.com/de-de/library/office/jj220060.aspx

15.9.2 Typen von Office-Add-ins

Es gibt drei Typen von Office-Add-ins, die man je nach ihrem „Einsatzort" unterscheidet:

- **Aufgabenbereich-Add-in:** Dieses Office-Add-in zeigt sich im Aufgabenbereich der Office-Anwendung am rechten Rand des Dokumentfensters. Add-ins dieser Art eignen sich insbesondere dazu, dem Anwender kontextbezogene Informationen und Funktionen – für die Suche oder das Übersetzen von Inhalten beispielsweise – zu liefern.
- **Inhalts-Add-in:** Dieses Office-Add-in erscheint ähnlich wie ein Excel-Diagramm als abgegrenzter Bereich innerhalb des Dokuments. Inhalts-Add-ins eignen sich vor allem für die Visualisierung von Daten oder die Wiedergabe von YouTube-Videos und anderen Internetmedien.
- **Mail-Add-in:** Es 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-Add-ins funktionieren nur im Zusammenspiel mit Microsoft Exchange ab Version 2013, normale POP- und IMAP-Konten werden nicht unterstützt.

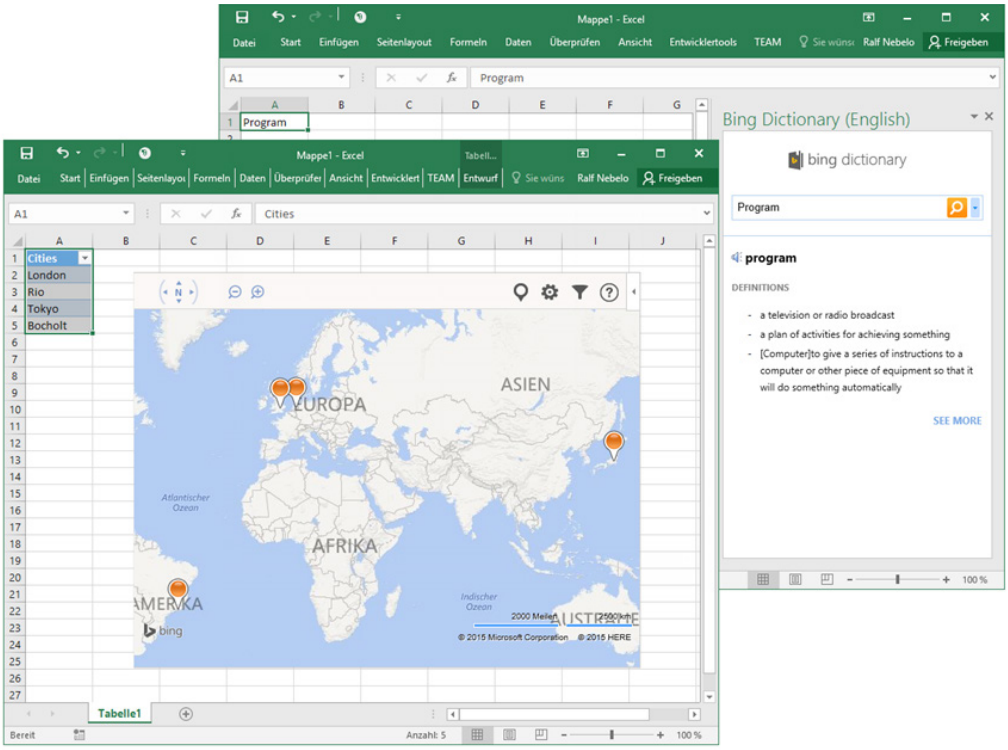


BILD 15.26: Microsofts Suchmaschine Bing stellt Excel-Anwendern unter anderem ein Aufgabenbereich-Add-in (hinten) sowie ein Inhalts-Add-in zur Verfügung.

Während Mail-Add-ins quasi naturgemäß auf Outlook festgelegt sind, sollten Inhalts- und Aufgabenbereich-Add-ins von der Konzeption her eigentlich in allen Office-Anwendungen nutzbar sein, die der Bearbeitung von Dokumenten dienen. Momentan unterstützen aber nur Excel und PowerPoint 2016 beide Add-in-Typen, während sich Project und Word nur mit Aufgabenbereich-Add-ins erweitern lassen. Die Tabelle zeigt den aktuellen Stand, der sich jedoch jederzeit ändern kann.

Office-Anwendung	Inhalts-Add-ins	Mail-Add-ins	Aufgabenbereich-Add-ins
Access 2016	-	-	-
Access-Web-App	+	-	-
Excel 2016	+	-	+
Excel-Web-App	+	-	+
Excel für iPad	+	-	+
Outlook 2016	-	+	-
Outlook für Mac	-	+	-
Outlook-Web-App	-	+	-
PowerPoint 2016	+	-	+

Office-Anwendung	Inhalts-Add-ins	Mail-Add-ins	Aufgabenbereich-Add-ins
PowerPoint-Web-App	+	-	+
Project 2016	-	-	+
Word 2016	-	-	+
Word-Web-App	-	-	+
Word für iPad	-	-	+

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

15.9.3 Werkzeuge für die Entwicklung von Office-Add-ins

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

Allerdings sollte man es mit der Bescheidenheit 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 eines Office-Add-ins ist *Visual Studio (ab Version 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 Office-Add-in-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 unserem ersten Beispiel-Add-in 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 für Speicherortangaben ausschließlich echte URLs (ohne „file:///“-Präfix) akzeptiert.

Fügen Sie der Netzwerkfreigabe zunächst ein neues Stammverzeichnis für die Add-in-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 des Beispiel-Add-ins bildet.

Die Webseite von SimpleApp

Zunächst erstellen Sie die Webseite des Beispiel-Add-ins, indem Sie eine HTML-Datei mit folgendem Inhalt anlegen (die erste Zeile bitte weglassen, sie verweist nur auf den Speicherort der Datei bei den Download-Dateien zum Buch). 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 Add-in-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 den Download-Dateien zum Buch 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 unseres Beispiel-Add-ins 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/1.1"
  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="Mein erstes Office-Add-in"/>
  <IconUrl DefaultValue=
    "\\NAS-Archive\Public\OfficeApps\SimpleApp\SimpleApp.png"/>
  <Hosts>
    <Host Name="Document"/>
    <Host Name="Workbook"/>
  </Hosts>
  <DefaultSettings>
    <SourceLocation DefaultValue=
      "\\NAS-Archive\Public\OfficeApps\SimpleApp\SimpleApp.html"/>
  </DefaultSettings>
  <Permissions>ReadWriteDocument</Permissions>
</OfficeApp>
```

Speichern Sie diese Manifestdatei unter dem Namen *SimpleApp.xml* im Stammverzeichnis (respektive Manifestordner) *OfficeApps*.

Hinweis

Im Normalfall werden Manifestdateien *nicht* lokal gespeichert. Das ist nur für die Dauer der Programmierung der Fall oder wenn das Office-Add-in ausschließlich für Teilnehmer des eigenen Heim- oder Firmennetzwerks gedacht ist. Möchte man ein Office-Add-in in Microsofts eingangs erwähntem Office Store veröffentlichen (und es damit weltweit für *alle* Office-Anwender verfügbar machen), so leitet man das Manifest im Rahmen eines offiziellen Anmelde- und Prüfverfahrens an Microsoft weiter. Informationen darüber finden Sie hier *[Link 41]*:

https://msdn.microsoft.com/de-DE/library/office/fp123515

Die HTML-Datei und alle weiteren Bestandteile der App gehören dann auf einen öffentlich erreichbaren Webserver, dessen Adresse im Manifest zu vermerken ist.

Der konstruktive Inhalt der abgedruckten Manifestdatei beginnt mit einem XML-Element namens *OfficeApp*, das neben dem verwendeten XML-Schema gleich auch den Add-in-Typus definiert. Letzteres fällt in die Zuständigkeit des Attributs *xsi:type*, das im Falle eines Aufgabenbereich-Add-ins den Wert „TaskPaneApp“ enthält. Bei einem Inhalts- oder Mail-Add-in würden die Attributwerte „ContentApp“ beziehungsweise „MailApp“ lauten.

Das *Id*-Element benennt die GUID der App, die man als eine Art Seriennummer bezeichnen könnte. Diese sollte für jede App neu generiert werden, was sich unter anderem mit dem *Online GUID Generator* (www.guidgenerator.com, *[Link 42]*) erledigen lässt. Im jeweiligen Verbreitungsraum der App (Office Store oder Netzwerk) muss die GUID auf jeden Fall eindeutig sein. Ist sie das nicht, taucht die App gar nicht erst im Auswahldialog der Office-Anwendung auf. Weniger kritisch ist der Inhalt des *Version*-Elements, das die Versionsnummer des Codegebildes angibt.

Die XML-Elemente *DisplayName* und *Description* definieren den Namen der App sowie eine Beschreibung ihrer Funktion, der Name des Entwicklers lässt sich im Element *ProviderName* verewigen. *IconUrl* verweist auf den Speicherort der Icon-Datei. Dabei handelt es sich im konkreten Fall um die Projektdatei *SimpleApp.png*, die Sie von den Download-Dateien zum Buch in den Projektordner *SimpleApp* kopieren sollten.

Hinweis

Die Icon-Datei wird nur sichtbar, wenn Sie die hinter *DefaultValue* angegebene URL an Ihre konkreten Gegebenheiten anpassen. Dabei müssen Sie stets absolute Angaben machen; relative URLs wie in der Webseitendatei sind in der Manifestdatei nicht zulässig.

Im Abschnitt *Hosts* bestimmt das Manifest, mit welchen Office-Dokumenten die App zusammenarbeiten kann. Dabei steht „Document“ für ein Word-Dokument, „Workbook“ für eine Excel-Arbeitsmappe, „Project“ für ein Microsoft-Project-Projekt und „MailBox“ für ein Outlook-Postfach.

Das Element *Permissions* weist der App die Rechte für den Umgang mit den aufgeführten Office-Dokumenten zu. Der Wert *ReadWriteDocument* erlaubt hier Lese- und Schreibzugriffe und bildet das Maximum an zuweisbaren Rechten. Weitere Abstufungen wären *WriteDocument* (nur Schreiben), *ReadDocument* (nur Lesen) und *Restricted*, die dem Office-Add-in gar keinen Zugriff gewährt.

Das Element *SourceLocation* im Abschnitt *DefaultSettings* schließlich verweist auf den Speicherort der Add-in-eigenen Webseite. Auch hier müssen Sie die angegebene (absolute!) URL an Ihre Gegebenheiten anpassen.

Vertrauenswürdige Add-in-Kataloge

Die Office-Anwendungen laden lokal gespeicherte Office-Add-ins grundsätzlich nur aus einem „Vertrauenswürdigen Add-in-Katalog“, wobei es sich um den Ordner handelt, in dem die Manifestdateien gespeichert sind. Im Fall unseres Beispiel-Add-ins wäre das also der Ordner *OfficeApps*.

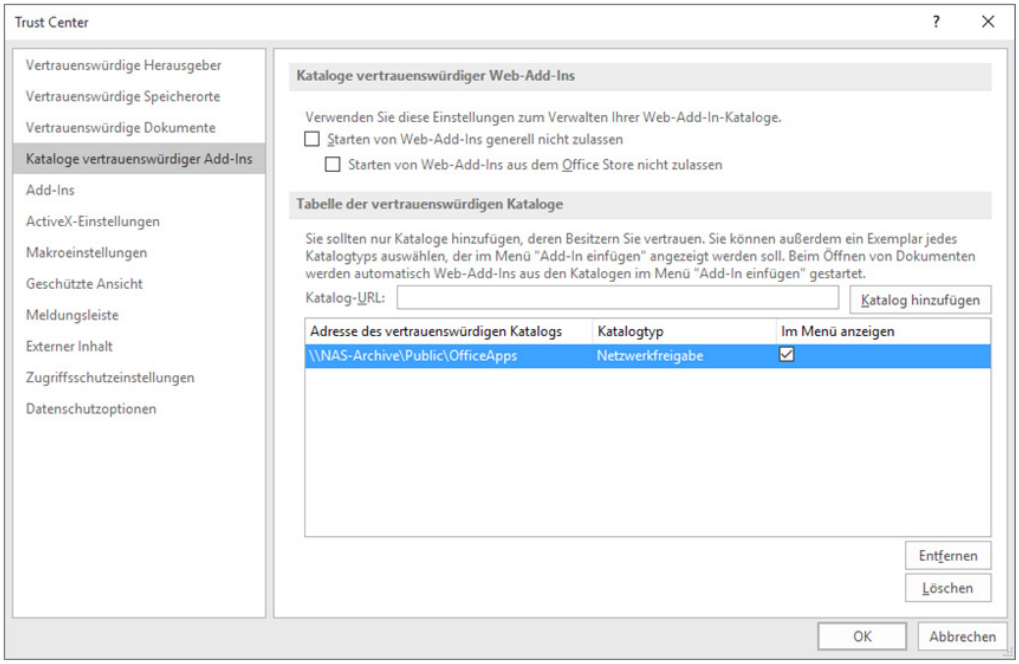


BILD 15.27: Lokal gespeicherte Apps stehen nur zur Auswahl, wenn man deren Manifestordner zum „Vertrauenswürdigen Katalog“ erklärt.

Um diesen Ordner in den Status der Vertrauenswürdigkeit zu erheben, starten Sie Excel 2016 und wählen Datei | Optionen. Klicken Sie links auf Trust Center und danach auf die Schaltfläche Einstellungen für das Trust Center. Markieren Sie links Kataloge vertrauenswürdiger Add-ins, und tragen Sie den UNC-Pfad des Ordners in das Textfeld „Katalog-URL“ ein – er sollte die Form „\\server\freigabe\manifestordner“ haben. Im Fall unseres Entwicklungsrechners lautet der Pfad:

\\NAS-Archive\Public\OfficeApps

Unnötig zu erwähnen, dass Sie auch diesen Pfad Ihren Gegebenheiten entsprechend ändern müssen. Wählen Sie dann KATALOG HINZUFÜGEN. Das Listenfeld enthält jetzt einen neuen Eintrag mit einem Kontrollkästchen dahinter („Im Menü anzeigen“). Das müssen Sie unbedingt einschalten, da Ihre Apps ansonsten nicht im Auswahldialog erscheinen. Anschließend schließen Sie alle Dialoge mit OK und starten Excel neu.

Add-in-Start

Das Beispiel-Add-in ist jetzt betriebsbereit und kann zum ersten Mal gestartet werden. Wählen Sie EINFÜGEN | MEINE ADD-INS, wobei Sie *nicht* auf das Icon, sondern auf dessen Pfeilsymbol klicken. Nach einem Klick auf ALLE ANZEIGEN und dann auf FREIGEGEBENE ORDNER sollte das Dialogfeld einen Eintrag namens „SimpleApp“ enthalten, dessen Icon aus einem Smiley besteht.

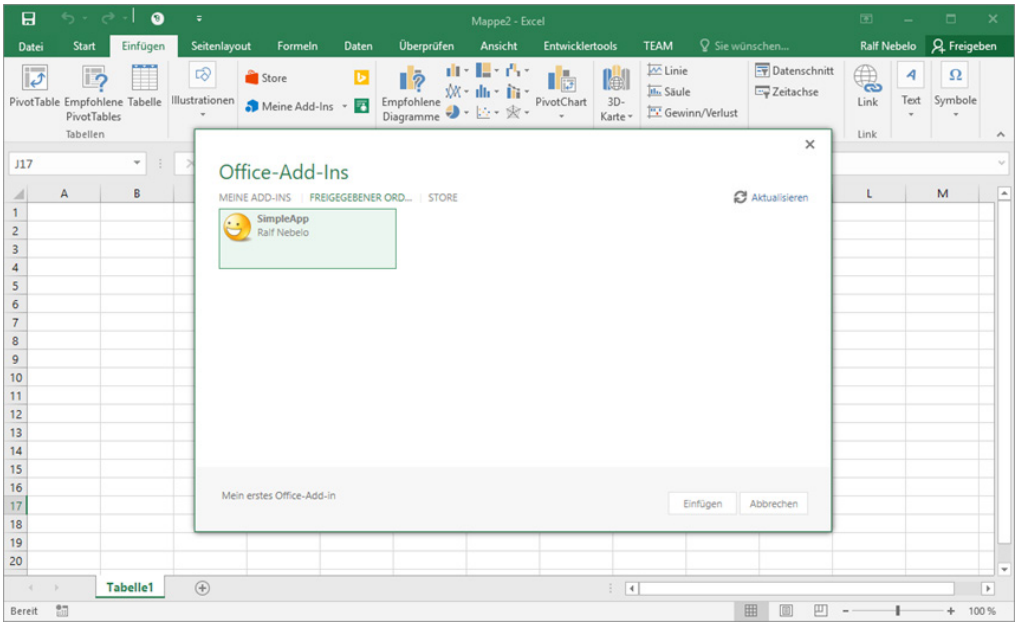


BILD 15.28: Wenn bis hierher alles geklappt hat, steht das Beispiel-Add-in nun im Auswahldialog bereit.

Fehlt der Eintrag, empfiehlt sich ein Klick auf die Schaltfläche AKTUALISIEREN, um die Anzeige auf den neuesten Stand zu bringen. Der Rest ist simpel: Markieren Sie „SimpleApp“, und wählen Sie EINFÜGEN.

Wenn alles geklappt hat, erscheint das Office-Add-in nun am rechten Rand des Excel-Fensters. Inhaltlich bietet es nicht viel: Sie können eine Webadresse eingeben und die entsprechende Seite dann mit einem Klick auf den Button in Ihrem Standard-Browser öffnen. SimpleApp halt!

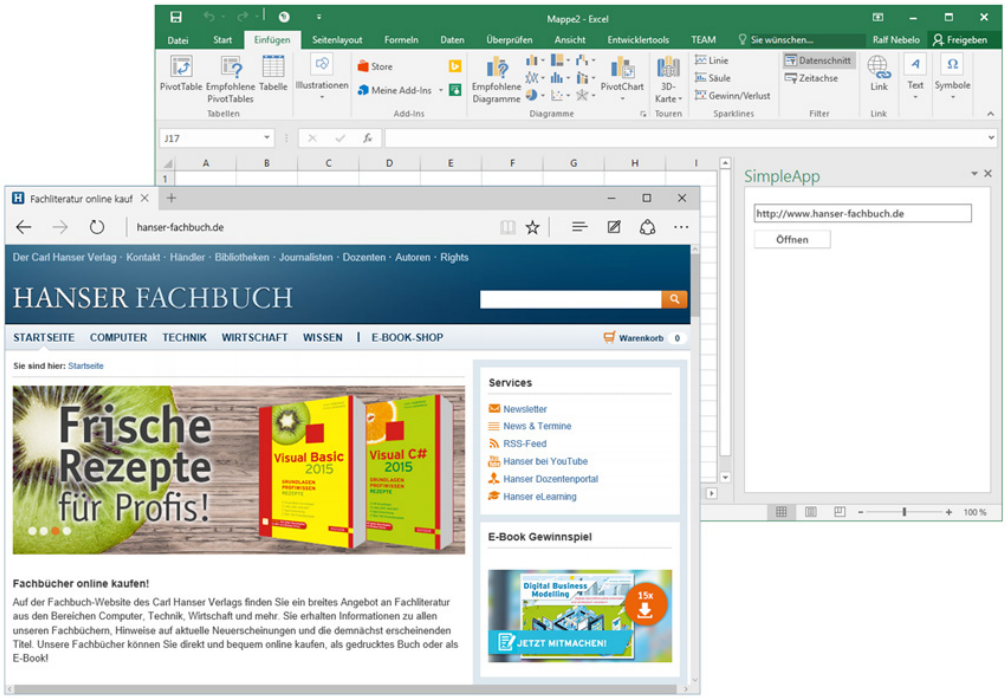
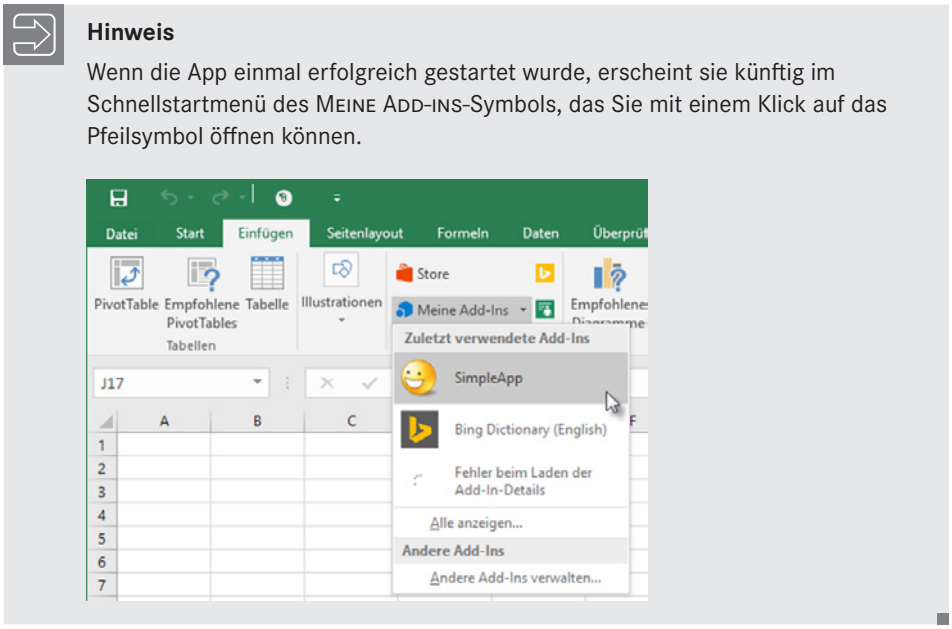


BILD 15.29: Das Beispielprojekt SimpleApp in Aktion



Hinweis

Wenn die App einmal erfolgreich gestartet wurde, erscheint sie künftig im Schnellstartmenü des Meine Add-ins-Symbols, das Sie mit einem Klick auf das Pfeilsymbol öffnen können.

15.9.5 Das JavaScript-API für Office

SimpleApp ist zwar schon ein „echtes“ Office-Add-in, hat mit Office-Programmierung aber noch nicht viel zu tun. Es ist absolut isoliert und kann anders als jedes VBA-Makro weder mit der Host-Anwendung (im konkreten Fall also Excel) kommunizieren noch Daten mit dem aktuellen Dokument austauschen.

Für diese Formen der Interaktion benötigt man das *JavaScript-API für Office*. Dabei handelt es sich um eine relativ umfangreiche Funktionsbibliothek, die mittlerweile in der Version 1.1 vorliegt und auf einem Microsoft-Server online verfügbar ist (natürlich kann man die JavaScript-Datei auch herunterladen und lokal abspeichern). Um diese Bibliothek nutzen zu können, muss man sie durch ein (weiteres) *script*-Element im HTML-Kopf der Add-in-Webseite anmelden:

```
<script src=
  "https://appsforoffice.microsoft.com/lib/1/hosted/Office.js"
  type="text/javascript">
</script>
```

API-Anatomie

Das JavaScript-API stellt dem Office-Add-in ein Objektmodell für den programmierten Zugriff auf die Host-Anwendung und den Datenaustausch mit dem Dokument zur Verfügung. Eine vollständige Dokumentation finden Sie hier *[Link 43]*:

http://msdn.microsoft.com/library/fp142185.aspx

An der Spitze des Objektmodells steht das *Office*-Objekt. Es repräsentiert das API und verweist mit seiner *context*-Eigenschaft auf ein gleichnamiges Objekt, das den Laufzeitkontext des Office-Add-ins verkörpert. Dieses Objekt wiederum stellt die Verbindung zum *Document*-Objekt her, welches Zugriff auf das konkrete Dokument gewährt und bei Aufgabenbereich- oder Inhalts-Add-ins stets im Mittelpunkt des Interesses steht.

Bevor das Objektmodell nutzbar ist, muss das API initialisiert werden. Dazu setzt man die folgende Anweisung an den Anfang der Skriptdatei:

```
Office.initialize = function (reason) {
}
```

Die Zeile installiert zugleich einen Ereignis-Handler, der beim Laden des Office-Add-ins ausgeführt wird. Den Handler kann man mit Code füllen, um eigene Initialisierungen vorzunehmen oder auf unterschiedliche Startbedingungen zu reagieren. So könnte es etwa von Interesse sein, ob das Add-in erstmals eingefügt wurde oder bereits im Dokument vorhanden war. Zur Beantwortung der Frage sollte man den von Office übergebenen *reason*-Parameter auswerten, was folgendermaßen aussehen kann:

```
Office.initialize = function (reason) {
  if (reason == "inserted")
    showText(„Add-in wurde neu eingefuegt");
  if (reason == "documentOpened")
    showText("Add-in wurde mit Dokument geoeffnet");
```

```
}
function showText(text) {
  document.getElementById("TextBox1").value = text;
}
```

Auf Dokumentereignisse reagieren

Alternativ lässt sich die *Office.initialize*-Routine auch nutzen, um einen Handler für ein Dokumentenereignis – die Änderung der Auswahl beispielsweise – einzurichten. Dazu benutzt man die *addHandlerAsync*-Methode des *Document*-Objekts und übergibt ihr eine *Office.EventType*-Konstante, die das gewünschte Ereignis definiert, sowie den Namen der Routine, die das Ereignis behandeln soll. Das Beispiel

```
Office.initialize = function (reason) {
  Office.context.document.addHandlerAsync(
    Office.EventType.DocumentSelectionChanged, myHandler);
}
function myHandler(eventArgs) {
  showText(„Auswahl geaendert");
}
```

erklärt die Routine *myHandler* zum offiziellen Event-Handler für das *DocumentSelectionChanged*-Ereignis. Das tritt in Excel auf, sobald der Anwender eine andere Zelle innerhalb des Arbeitsblatts aktiviert. In Word wirkt sowohl das Verschieben der Schreibmarke als auch das Markieren von Inhalten als Event-Auslöser.

Rückgabe von markiertem Text

Wer nicht nur auf eine Änderung der Auswahl reagieren, sondern auch auf deren Inhalt zugreifen will, verwendet die *getSelectedDataAsync*-Methode des *Document*-Objekts und übergibt ihr eine passende *Office.CoercionType*-Konstante.

Die bestimmt den sogenannten Koersionstyp, das ist der Datentyp, den die Methode zurückgeben soll. Dabei kann es sich im einfachsten Fall um den reinen Text der Auswahl handeln, der sich wie folgt ermitteln lässt:

```
Office.context.document.getSelectedDataAsync(
  Office.CoercionType.Text, function(result) {
    if (result.status == "succeeded")
      showText(result.value);
    else
      showText(result.error.message);
  }
);
```

Die Methode greift auf den Inhalt der Auswahl zu und retourniert das Objekt *result* an die eingeschlossene *function*-Routine. Die überprüft zunächst die *status*-Eigenschaft des Objekts. Enthält diese den Text „succeeded“, war der Abruf des Auswahltextes erfolgreich. Eine leere *status*-Variable oder der Text „failed“ deuten auf einen Fehler hin, dessen Beschreibung dann in *result.error.message* steckt.

Rückgabe von mehreren markierten Werten

Neben simplem Text unterstützt das JavaScript-API noch weitere Rückgabetypen: *Matrix* und *Table* beispielsweise. Die eignen sich für die Rückgabe respektive das Festlegen von tabellarischen Daten, wobei eine Table Kopfzeilen enthalten kann, eine Matrix nicht.

Die drei bislang genannten Typen funktionieren in allen Add-in-tauglichen Office-Anwendungen. Table- oder Matrix-Daten stehen also nicht nur in Excel zur Verfügung, was man erwarten würde, sondern beispielsweise auch in Word, wo es ja ebenfalls Tabellen gibt.

Das JavaScript-API wandelt Daten, die nicht im passenden Typ vorliegen, in den meisten Fällen automatisch um. Sollte der Entwickler also einen Text anfordern, der Anwender aber einen Bereich mit mehreren Tabellenzellen markiert haben, so liefert das API dennoch einen Text zurück – in Form eines tabulatorseparierten Strings nämlich, der die Werte aller ausgewählten Zellen enthält.

Wer keinen Sammel-String, sondern jeden markierten Zellwert einzeln verarbeiten möchte, fordert den Typ Matrix beim Aufruf der *getSelectedDataAsync*-Methode ausdrücklich an:

```
Office.context.document.getSelectedDataAsync(
  Office.CoercionType.Matrix, function(result){
    if (result.status == "succeeded")
      showData(result.value);
  }
);
```

Die *value*-Eigenschaft des *result*-Objekts enthält dann im Erfolgsfall ein zweidimensionales Array mit den markierten Zellwerten. Die kann die aufgerufene *showData*-Funktion wie folgt mit zwei verschachtelten *for*-Schleifen zur Anzeige bringen:

```
function showData(data) {
  var text = "";
  for (var y = 0 ; y < data.length; y++) {
    for (var x = 0; x < data[y].length; x++) {
      text += data[y][x] + ", ";
    }
  }
  document.getElementById("TextBox1").value = text;
}
```

Markierte Dokumentinhalte ändern

Soll das Office-Add-in den Inhalt der Auswahl nicht nur lesen, sondern aktiv verändern, wählt man die *setSelectedDataAsync*-Methode und übergibt ihr im einfachsten Fall einen Text, der die Dokumentauswahl ersetzt:

```
Office.context.document.setSelectedDataAsync("Neuer Text");
```

Im Fall von Word ersetzt der Text den Inhalt der Markierung, im Fall von Excel wird stets der komplette Inhalt der aktiven Zelle ausgetauscht, auch wenn der Anwender nur einen Teil davon markiert hat.

Soll die App die Inhalte *mehrerer* Zellen ändern, die Teil der aktuellen Auswahl sind, definiert man zunächst ein passendes Array, das die neuen Werte enthält:

```
var arr = [['a', 'b'],['c', 'd'],['e', 'f']];
```

Dieses Array leitet man dann nebst einem weiteren Argument zur Festlegung des Datentyps Matrix an die *setSelectedDataAsync*-Methode weiter. Das Beispiel

```
Office.context.document.setSelectedDataAsync
  (arr, {coercionType: 'matrix'});
```

füllt einen zwei Spalten breiten und drei Zeilen hohen Tabellenbereich, dessen obere linke Ecke die aktive Zelle bildet, mit den Buchstaben „a“ bis „f“.

Das deutlich spektakulärere Einfügen von Bildern gelingt ausschließlich im Office-eigenen Textprogramm Word. In Excel ist es bislang nicht vorgesehen.

Entwicklerdefinierte Dokumentinhalte ändern

Über die vom Anwender festgelegte Auswahl hinaus können Office-Add-ins auch mit Regionen eines Dokuments interagieren, die der Entwickler bestimmt. Dazu muss die Region allerdings einen eindeutigen Namen besitzen, was im Fall von Excel auf benannte oder als Tabelle/Liste formatierte Arbeitsblattbereiche, in Word unter anderem auf Inhaltssteuerelemente zutrifft.

Für den Zugriff auf eine solche Region generiert die App zunächst eine „Bindung“, über die sie dann den Inhalt der Region sowohl lesen als auch ändern kann. Dazu stehen ihr die Methoden *getDataAsync* und *setDataAsync* zur Verfügung, die ähnlich funktionieren wie die zuvor vorgestellten Methoden *getSelectedDataAsync* und *setSelectedDataAsync* für den Umgang mit Markierungsinhalten. Die verfügbaren Datentypen sind in beiden Fällen identisch.

Die zuvor schon erwähnte JavaScript-API-Dokumentation *[Link 43]* enthält ausführliche Infos über den Zugriff auf Regionen.

Auf die Dokumentdatei zugreifen

Neben der Interaktion mit markierten Dokumentinhalten und benannten Regionen erlaubt das JavaScript-API auch einen programmierten Zugriff auf die Dokumentdatei. Den erhält man mit Hilfe der *getFileAsync*-Methode. Damit kann man das Dokument beispielsweise versenden oder im Firmennetzwerk veröffentlichen. Zugang zum gesamten Dokumentinhalt liefert die Methode aber nicht.

Auch hier müssen wir Sie auf die JavaScript-API-Dokumentation *[Link 43]* verweisen. Eine ausführliche Behandlung des Themas würde den Rahmen einer Einführung sprengen.

15.9.6 Beispiel 2: ComplexApp

Ein zweites Beispielprojekt soll den Einsatz des JavaScript-API und den dadurch ermöglichten Datenaustausch zwischen Anwendung und App demonstrieren. Es übernimmt einen in Celsius angegebenen Temperaturwert aus der aktuellen Zelle und rechnet diesen wahlweise in Kelvin oder Fahrenheit um. Ein Mausklick fügt das Ergebnis dann wieder in die aktuelle Zelle ein.

Wenn Sie dieses Beispielprojekt praktisch nachvollziehen möchten, fügen Sie dem *OfficeApps*-Stammordner zunächst einen neuen Unterordner namens *ComplexApp* hinzu. Dort speichern Sie die folgende HTML-Datei, die die Webseite der App bildet, unter dem Namen *ComplexApp.html* ab:

```
<! 15\OfficeApps\ComplexApp\ComplexApp.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=
      "https://appsforoffice.microsoft.com/lib/1/hosted
      /Office.js" type="text/javascript"></script>
    <script src="ComplexApp.js"></script>
  </head>
  <body>
    <div>
      Aktuelle Auswahl: <br>
      <input type="text" value="" id="txtValue"
        style="width: 254px" />
      <input type="button" value="Celsius in Kelvin"
        id="btnToKelvin" onclick="convertKelvin()"
        style="margin-top: 5px; width: 258px" />
      <input type="button" value="Celsius in Fahrenheit"
        id="btnToFahrenheit" onclick= „convertFahrenheit()"
        style="margin-top: 5px; width: 258px" />
      <hr width="258" align="left">
      Ergebnis: <br>
      <input type="text" value="" id="txtResult"
        style="width: 254px" />
      <input type="button" value="Ergebnis einfügen"
        id="btnInsert" onclick= "insertResult()"
        style="margin-top: 5px; width: 258px" />
    </div>
    <div>
      <hr width="258" align="left">
      Meldung: <br>
      <textarea id="txtMessage" rows="6"></textarea>
      <hr width="210" align="left">
    </div>
  </body>
</html>
```

Der *head*-Bereich der Webseite unterscheidet sich kaum von seinem Gegenstück in *SimpleApp.html*. Einziger Unterschied ist das zweite *script*-Element, das – wie zuvor schon beschrieben – den Verweis auf das JavaScript-API herstellt.

Innerhalb des *body*-Blocks sind die Unterschiede größer. Die Anweisungen dort fügen der Webseite gleich zwei Textboxen, drei Buttons und ein mehrzeiliges Textarea-Control aus dem HTML-Fundus hinzu. Die erste Textbox (id="txtValue") zeigt den Inhalt der aktuell markierten Zelle, die zweite (id="txtResult") das Ergebnis, das nach dem Klick auf die oberen beiden Buttons (id="btnToKelvin" und id="btnToFahrenheit") berechnet wird. Button Nummer Drei (id="btnInsert") überträgt das Ergebnis zurück in das Dokument. Das Textarea-Control schließlich (id="txtMessage") ist für die Anzeige möglicher Fehlermeldungen zuständig.

Die Skriptdatei von ComplexApp

Die Skriptdatei des Beispiel-Add-ins trägt den Namen *ComplexApp.js* und ist ebenfalls im Unterordner *ComplexApp* zu speichern.

```
/* 15\OfficeApps\ComplexApp\ComplexApp.js */
Office.initialize = function (reason) {
  Office.context.document.addHandlerAsync(
    Office.EventType.DocumentSelectionChanged, myHandler);
}

function myHandler(eventArgs) {
  showMessage("");
  Office.context.document.getSelectedDataAsync(
    Office.CoercionType.Text, function (result) {
    if (result.status == "succeeded")
      document.getElementById("txtValue").value = result.value;
    else
      showMessage(result.error.message);
  });
}

function convertKelvin() {
  var celsius =
    parseFloat(document.getElementById("txtValue").value)
  var kelvin = celsius + 273.15
  document.getElementById("txtResult").value = kelvin
}

function convertFahrenheit() {
  var celsius =
    parseFloat(document.getElementById("txtValue").value)
  var fahrenheit = (celsius * 1.8) + 32
  document.getElementById(„txtResult").value = fahrenheit
}

function insertResult() {
  showMessage("");
  var correctedResult =
```

```
    document.getElementById("txtResult").value.replace(".", ",")
  Office.context.document.setSelectedDataAsync(correctedResult,
    function (result) {
    if (result.status == "failed")
      showMessage(result.error.message);
  });
}

function showMessage(message) {
  document.getElementById("txtMessage").value = message;
}
```

Die *Office.initialize*-Zeile, die ja beim Laden des Office-Add-ins automatisch aufgerufen wird, erklärt die Funktion *myHandler* zum zuständigen Event-Handler, der bei jeder Änderung der Dokumentauswahl ausgeführt wird. Die Funktion ermittelt dann den Wert der aktuellen Zelle mit Hilfe der API-Funktion *getSelectedDataAsync* und trägt diesen in die Textbox mit der ID *txtValue* ein.

Das Anklicken des obersten Buttons fällt in die Zuständigkeit der *convertKelvin*-Routine. Diese liest den Wert der Textbox *txtValue*, addiert die Zahl 273,15 dazu und trägt das Ergebnis in die Textbox *txtResult* ein.

Bei *convertFahrenheit* verhält es sich ähnlich. Die Routine tritt beim Anklicken des zweiten Buttons von oben in Aktion. Sie liest ebenfalls den Wert der Textbox *txtValue*, multipliziert diesen aber mit 1,8 und rechnet 32 hinzu, ehe sie das Ergebnis in die Textbox *txtResult* überträgt.

Ein Klick auf den dritten Button schließlich ruft die *insertResult*-Routine auf den Plan. Diese ermittelt den momentanen Wert der Textbox *txtResult*, tauscht einen eventuell darin befindlichen Dezimalpunkt gegen das hierzulande übliche Dezimalkomma aus und überträgt das Ergebnis mit Hilfe der *setSelectedDataAsync*-Methode des JavaScript-API in die markierte Zelle.

Die Manifestdatei von ComplexApp

Die Manifestdatei unseres zweiten Beispiel-Add-ins hat folgenden Inhalt und ist unter dem Namen *ComplexApp.xml* im Stamm-/Manifestordner *OfficeApps* zu speichern. Wie schon im ersten Beispiel, so sind auch hier die verwendeten (absoluten) URLs, die auf die Speicherorte der Webseiten- und der Icon-Datei verweisen, an die eigenen Gegebenheiten anzupassen.

```
<! 15\OfficeApps\ComplexApp.xml >
<?xml version="1.0" encoding="utf-8"?>
<OfficeApp
  xmlns="http://schemas.microsoft.com/office/appforoffice/1.1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:type="TaskPaneApp">
  <Id>08afd7fe-1631-42f4-84f1-5ba51e242f12</Id>
  <Version>1.0</Version>
  <ProviderName>Ralf Nebelo</ProviderName>
  <DefaultLocale>EN-US</DefaultLocale>
```

```
  <DisplayName DefaultValue="ComplexApp"/>
  <Description DefaultValue="Mein zweites Office-Add-in"/>
  <IconUrl DefaultValue=
    "\\NAS-Archive\Public\OfficeApps\ComplexApp\ComplexApp.png"/>
  <Hosts>
    <Host Name="Document"/>
    <Host Name="Workbook"/>
  </Hosts>
  <DefaultSettings>
    <SourceLocation DefaultValue=
      "\\NAS-Archive\Public\OfficeApps\ComplexApp\ComplexApp.html"/>
  </DefaultSettings>
  <Permissions>ReadWriteDocument</Permissions>
</OfficeApp>
```

ComplexApp anzeigen und nutzen

Zum Starten des Beispiel-Add-ins wählen Sie Einfügen | Meine Add-ins, wobei Sie wiederum *nicht* auf das Icon, sondern auf dessen Pfeilsymbol klicken. Nach einem Klick auf Alle anzeigen und dann auf Freigegebene Ordner sollte das Dialogfeld einen Eintrag namens „ComplexApp“ enthalten, dessen Icon aus einem Stern besteht. Fehlt der Eintrag, klicken Sie auf Aktualisieren, um die Anzeige auf den neuesten Stand zu bringen. Anschließend markieren Sie „ComplexApp“ und wählen Einfügen.

Das Office-Add-in sollte nun am rechten Rand des Excel-Fensters erscheinen. Testen Sie es, indem Sie ein paar Temperaturwerte in das Arbeitsblatt schreiben und diese anschließend der Reihe nach anklicken. Der jeweilige Zellwert erscheint dann in der Textbox Aktuelle Auswahl der ComplexApp.

Nach einem Klick auf Celsius in Kelvin beziehungsweise Celsius in Fahrenheit sollte das Ergebnis der Umrechnung in der Textbox Ergebnis auftauchen, von wo aus Sie es mit einem Klick auf Ergebnis einfügen in die aktuelle Zelle übertragen können.

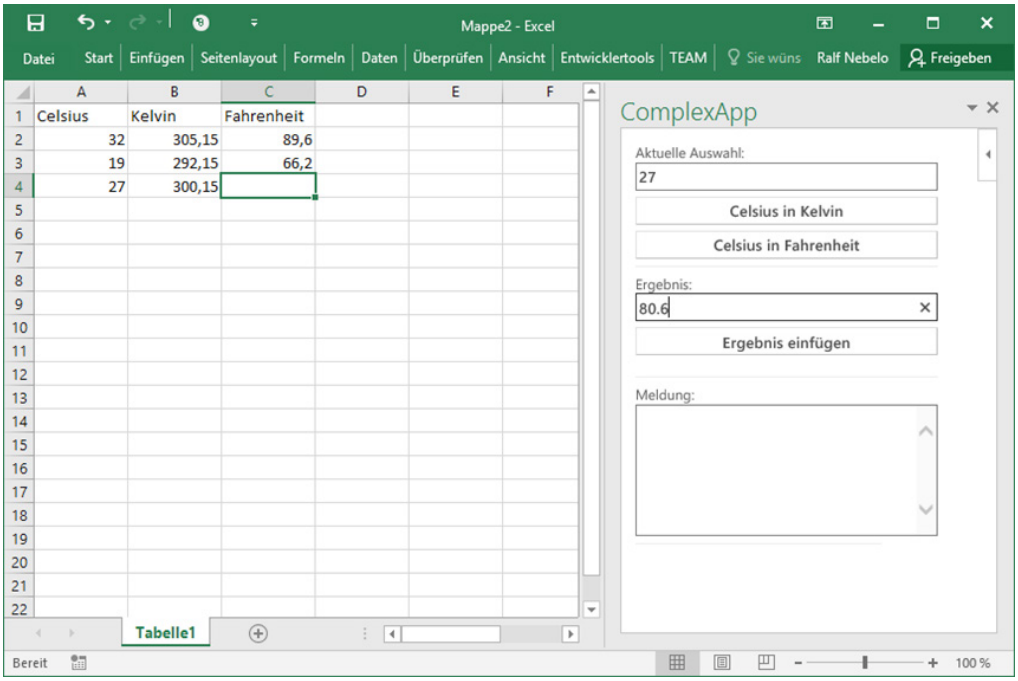


BILD 15.30: ComplexApp macht sich durchaus nützlich, indem sie Temperaturwerte von Celsius in Kelvin und Fahrenheit umrechnet.

Stichwortverzeichnis

B

G

M

N

T

U